Java

Basissyntax

- elke regel sluiten we af met een puntkomma
- we kunnen commentaar schrijven tussen de code
  - één regel → // *commentaar*
  - meerdere regels → /* *commentaar* */
- Java is hoofdlettergevoelig

Variabelen

= een plaats in het geheugen met een daaraan gekoppelde naam

- moeten gedeclareerd worden voor ze kunnen worden gebruikt:
  - datatype van de nieuwe variabele: **int**
  - naam van de nieuwe variabele: ***getal1***
- een waarde toekennen:
  - assignment-operator **=**
  - waarde ***18***

  → eerste keer toekennen = initialiseren

  → kan later opnieuw toegewezen worden

- verschillende datatypes
  - getallen (byte, short, int, long, float, double)
  - true/false (boolean)
  - tekens (char)
  - tekst (string)
  - verzameling van meerdere variabelen van hetzelfde datatype (array)

Access modifiers

bepaalt de mate waarin het attribuut, methode, … waar hij bijstaat toegankelijk is

- **public** (symbool **+**): wilt zeggen dat het toegankelijk is van eender waar
- **private** (symbool **-**): wilt zeggen dat het enkel toegankelijk is van binnen de klasse (of instanties)
- **package** (geen modifier; symbool **~**)
- **protected** (symbool **#**)

De thread

- de CPU voert opdrachten supersnel na mekaar uit (asynchroon, behalve bij multithreading)
- commando's worden één voor één uitgevoerd, van boven naar beneden
- als een commando uit meerdere expressies bestaat, worden deze uitgevoerd volgens de wiskundige precedentieregels

Operatoren

worden gebruikt om wiskundige, logische en vergelijkingsoperaties uit te voeren op variabelen

Rekenkundige operatoren

- geplaatst tussen 2 variabelen/constante waarden
- vormen samen een expressie die de computer zal uitrekenen

OPERATOR	VERKLARING	OPERATOR	VERKLARING
+	Optelling	**getal++**	getal = getal + 1
-	Aftrekking	**getal--**	getal = getal – 1
*****	Vermenigvuldiging	**getal1 += getal2**	getal1 = getal1 + getal2
/	Deling	**getal1 -= getal2**	getal1 = getal1 – getal2
%	Modulo (de rest van een deling)	**getal1 *= getal2**	getal1 = getal1 * getal2
-	Conversie van teken	**getal1 /= getal2**	getal1 = getal1 / getal2

Vergelijkingsoperatoren

- worden gebruikt om de inhoud van 2 variabelen of constante waarden te vergelijken
- geven altijd een boolean als resultaat

OPERATOR	VERKLARING	
==	Is gelijk aan	(werkt niet met strings → *variabele*.equals(‘*string*’))
<	Kleiner dan	
>	Groter dan	
<=	Kleiner dan of gelijk aan	
>=	Groter dan of gelijk aan	
!=	Verschillend van	

Logische operatoren

- worden gebruikt om de inhoud van booleans te vergelijken
- geven altijd een boolean als resultaat

OPERATOR	VERKLARING	
&&	Logische and	(enkel wanneer beide booleans true zijn krijg je true als uitkomst)
||	Logische or	(wanneer minstens één boolean true is, krijg je true als uitkomst)
!	Not (logische inversie)	(één boolean krijgt het omgekeerde van zijn waarde)

Toekenningsoperator

- wordt gebruikt om een waarde toe te kennen aan een variabele

OPERATOR	VERKLARING
=	Toekenningsoperator

Controlestructuren

if/else if/else

Een codeblok wordt uitgevoerd als de expressie *true* evalueert.

```
if(expressie) {
   codeblok
} else if(expressie) {
   codeblok
} else {
   codeblok
}
```

while

Herhaalt een codeblok zolang aan een bepaalde voorwaarde voldaan wordt.

```
while(voorwaarde) {codeblok}
```

do..while

Variant op de while loop; voert het codeblok eenmaal uit en herhaalt dat codeblok dan indien en zolang aan een bepaalde voorwaarde voldaan wordt.

```
do {codeblok} while(voorwaarde);
```

for

Voert het codeblok uit zo lang wordt voldaan aan de test-expressie.

```
for (init-expressie; test-expressie; update-expressie) {codeblok}
```

switch..case

De uitkomst van de expressie wordt vergeleken met elke *case*. Indien er een overeenkomst wordt gevonden wordt de bijhorende code uitgevoerd, anders wordt de *default* code uitgevoerd.

```
switch(expressie) {
    case n:
        codeblok
        break;
    case m:
        codeblok
        break;
    default:
        codeblok
}
```

Omzetten tussen datatypes

Van	Naar	Opdracht
string	int	`Integer.parseInt("string")`
string	float	`Float.parseFloat("string")`
string	double	`Double.parseDouble("string")`
int	string	`Integer.toString(integer)`
float	string	`Float.toString(float)`
double	string	`Double.toString(double)`

Methoden

= korte stukjes code die je kan aanroepen met een methodenaam
= een groep opdrachten die bij elkaar hoort en die een naam heeft

`public void resetFormulier()`	= <u>method header</u> `public` De access modifier `void` Het return type `resetformulier` De naam v.d. methode `(type naam)` Parameters (optioneel)

`{`
`txtVoornaam.setText("");`
`txtAchternaam.setText("");`
`txtAdres.setText("");`
`txtGemeente.setText("");`
	= <u>method body</u>
	→ de code die zal worden uitgevoerd wanneer de methode wordt aangeroepen
	→ de "implementatie" van de methode
	→ staat tussen 2 accolades

`return “voltooid”`	= <u>return statement</u> → als laatste regel van je methode → achter de return statement schrijf je de waarde die je wilt verzenden naar de locatie waar de methode werd aangeroepen → moet van hetzelfde type zijn als het return-type dat je in de method header hebt getypt → volgende zaken moeten v. hetzelfde type zijn ✓ het return type van de methode ✓ de waarde die je achter het return statement zet in de methode ✓ de variabele waarin je de return value "opvangt"
`}`	

- Je kan een methode aanroepen door de naam van de methode te typen met haakjes erachter
- Een methode mag ook zichzelf aanroepen in zijn implementatie (recursieve methode)

Verkorte notatie van methoden

bv. `+substring(beginIndex:int, eindIndex:int):String`

1. De **access modifier** (public is +, private is -)
2. De **naam van de methode** (hier: substring)
3. De **parameters** en hun datatypes, gescheiden door een komma.
4. Het **datatype van de returnwaarde** (*return type*), na de dubbelpunt

→ deze notatie bepaalt de **signatuur** van de methode volledig
- ✓ De naam van de methode
- ✓ De volgorde van de types van de parameters
- ✓ Het aantal parameters

Voordelen

- ✓ **Hergebruik van code**
  - minder typewerk
  - functionaliteit aanpassen moet slechts één keer op een centrale plaats
  - je mag eenzelfde methode vanop verschillende plaatsen in je code aanroepen

- ✓ **Complexiteit opsplitsen**
  - een moeilijk programma opsplitsen in kleinere, gemakkelijker op te lossen stappen
  - kleinere deelproblemen zijn gemakkelijker op te lossen dan één groot complex probleem

- ✓ **Complexiteit abstraheren**
  - We kunnen de methodes die deelproblemen oplossen ergens centraal oproepen, op die centrale plaats zien we een samenvatting van de functionaliteit die ons programma uitvoert

- ✓ **Vermijden dat stukken code gekopieerd/gedupliceerd worden**
  - Het dupliceren/kopiëren (copy-paste) van stukken code is NOOIT een goed idee
  - Het verhoogt de onderhoudskost van de applicatie
    (Als er iets in het gekopieerde stuk code moet veranderen, moet je dit overal veranderen waar je de code hebt gekopieerd. De kans dat je een stuk vergeet is groot)
  - Het verlaagt de overzichtelijkheid van de applicatie
    (uw code wordt veel langer door het kopiëren van grote stukken code)

- ✓ **Gemakkelijker samenwerken met andere programmeurs**
  - Indien je functionaliteit in je programma wenst die je zelf nog niet kan maken, kan je methodes die door andere programmeurs werden aangemaakt aanroepen
  - Jij moet je niets aantrekken van de manier waarop dit deelprobleem in de aangeroepen methode wordt opgelost, maar je bent er wel zeker van dat het wordt opgelost

Overloading

= methodes met dezelfde naam, maar met een verschillende signatuur (m.a.w. de volgorde van de parametertypes en/of het aantal parameters verschilt)

Voorbeeld

```
public double berekenVolumeCilinder(double straal) {
   return berekenVolumeCilinder(straal, 1);
}

public double berekenVolumeCilinder(double straal, double hoogte) {
   return straal * straal * 3.14 * hoogte;
}
```

Swing

Checkbox

vinkje (met bijbehorende tekst) dat de gebruiker kan aan- of uitzetten

→ de naam laten we beginnen met **chk** (bv. chkStatus)

Property	Getter	Omschrijving
selected	isSelected	Bepaalt of het vinkje al dan niet is aangeduid
enabled	isEnabled	Bepaalt of de staat van het vinkje veranderd kan worden
text	getText	Geeft de tekst die bij de checkbox hoort

Radio button

laat de gebruiker een keuze maken uit 2 of meer opties (bestaat uit een bolletje met bijbehorende tekst)

→ de naam laten we beginnen met **rbt** (bv. rbtKeuze)

Property	Getter	Omschrijving
selected	isSelected	Bepaalt of de radio button al dan niet is aangeduid
enabled	isEnabled	Bepaalt of de staat van de radio button veranderd kan worden
text	getText	Geeft de tekst die bij de radio button hoort

Buttongroup

een onzichtbare component die radio buttons groepeert opdat er maar één tegelijk geselecteerd kan zijn

→ de naam laten we beginnen met **btngrp** (bv. btngrpKeuze)

Debugger

= laat toe om de uitvoering van onze programma's te pauzeren en bepaalde tools te gebruiken om de verschillende omgevingsvariabelen waarmee ons programma op dat moment werkt te analyseren

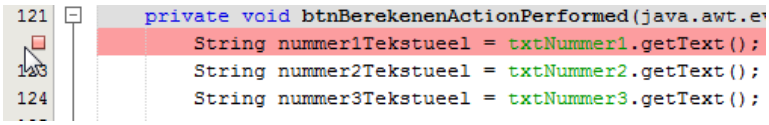
- een veelvoorkomende ondersteunende feature van een professionele IDE
- wordt gebruikt om:
  - ✓ Te leren in welke volgorde code wordt uitgevoerd (*thread*)
  - ✓ Te leren welke variabelen er op een bepaald moment tijdens de uitvoering van het programma gekend zijn + welke waarde ze op dat moment hebben.
  - ✓ Fouten die tijdens het uitvoeren van het programma voorkomen op te sporen

De debugger gebruiken

Breakpoints maken

door in de marge (links) te klikken naast de betreffende regel code waar je een breakpoint wilt

- → Je ziet een rood vierkantje verschijnen
- → Daarnaast wordt de betreffende regel code ook rood gemarkeerd
- → Je kan een breakpoint terug verwijderen door op het rode vierkantje te klikken

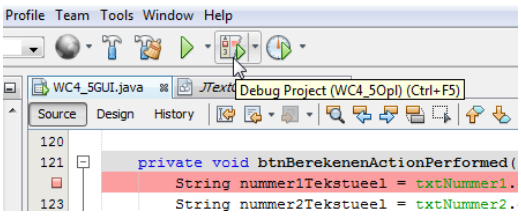


Stel de *main class* van je project correct in

1. Rechtsklik op de projectnaam in de *project explorer*.
2. Properties > Run > Main Class > Browse…
3. Kies de jFrame die je hebt aangemaakt

Ons programma starten in debug mode.

Druk op het icoon waarop een klein groen pijltje en een breakpoint staat afgebeeld

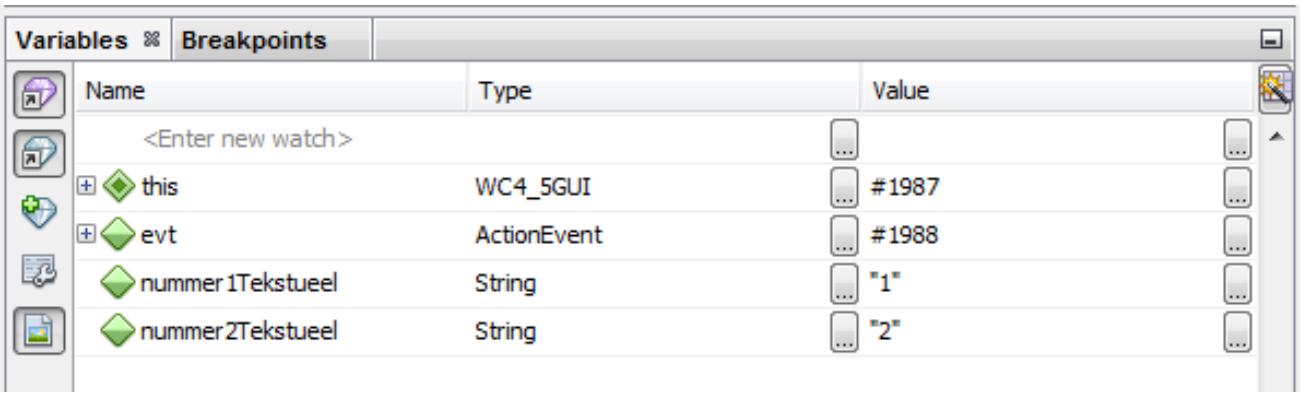


Besturing

Je kan de uitvoering van het programma besturen met de volgende knoppen:

1. **Stop**: De uitvoering van het programma (en de debugger) stoppen
2. **Pauze**: De uitvoering van het programma pauzeren
3. **Continue**: je gepauzeerde programma vanaf de huidige opdracht terug laten lopen tot het volgende breakpoint dat de thread tegen komt
4. **Step over**: de eerstvolgende opdracht (in groen gemarkeerd) wordt uitgevoerd
5. **Step over expression**
6. **Step into**: de eerstvolgende opdracht (in groen gemarkeerd) wordt uitgevoerd, maar wanneer op de groen gemarkeerde lijn code een methode wordt aangeroepen, pauzeert de thread op de eerste lijn code van de aangeroepen methode.
7. **Step out**: wanneer de thread binnen een methode zit, en we willen deze niet meer stap voor stap tot het einde van de body doorlopen kan je deze knop gebruiken.

Variabelenvenster



= een lijst van de variabelen die op dat moment (waar de thread nu staat) gedeclareerd zijn in je programma + de waarde die aan elk van die variabelen werd toegekend

Variabelen uitlezen via hover

Wanneer je hovert over de naam van een bep. variabele, komt zijn huidige waarde tevoorschijn.

Fouten

Er kunnen verschillende soorten fouten optreden in een programma

Functionele fouten

Een programma kan fouten bevatten en toch technisch perfect werken (het programma doet gewoon niet wat je verwacht dat het zal doen)

Technische fouten

Compile time fouten

Fouten die verhinderen dat een programma kan gecompiled worden

- → het programma kan dus niet starten
- → herkenbaar door een rode lijn onder de code wanneer je een programma schrijft
- → kunnen veel oorzaken hebben
  - ✓ syntaxfouten (bv. puntkomma's vergeten, haakjes vergeten af te sluiten)
  - ✓ ongeldige acties in de code

Runtime fouten

Fouten die niet kunnen gedetecteerd worden bij het compilen van het programma

- → Wanneer deze situatie zich voordoet bij de uitvoering van het programma krijgen we een zogenaamde *exception* (wordt zichtbaar in Netbeans als een hoeveelheid rode tekst die verschijnt in de console)
- → kan opgelost worden door de fout te reproduceren en er stap voor stap door te gaan m.d. debugger

Klasses

= een code-template op basis waarvan we instanties (*objecten*) kunnen maken

- → een nieuwe instantie maken van een klasse = de klasse instantiëren
- → je kan zo veel instanties maken van een klasse als je zelf wilt
- → bepaalt de volgende zaken voor de objecten die op basis van deze template worden gemaakt:
  - ✓ welke data deze objecten moeten kunnen opslaan
  - ✓ welke functionaliteiten deze objecten moeten hebben
- → elke klasse heeft een naam
  - is zo goed mogelijk een weerspiegeling v.d. data of functionaliteit die de klasse zal bevatten
  - is in PascalCase (UpperCamelCase) geschreven
- → voor elke klasse die we willen maken, zullen we een nieuw .java bestand aanmaken in ons project

`public class NaamKlasse`	= <u>class header</u> `public` De access modifier `class` Een vast keyword `NaamKlasse` De naam v.d. klasse

`{`
`  txtVoornaam.setText("");`
`  txtAchternaam.setText("");`
`  txtAdres.setText("");`
`  txtGemeente.setText("");`
`}`
	= <u>class body</u>
	→ welke data de objecten die op basis van deze klasse worden gemaakt moeten kunnen opslaan
	→ welke functionaliteiten de objecten die op basis van deze klasse worden gemaakt moeten hebben

Attributen

= een variabele waarvan de declaratie in de body van een klasse staat (≠ methode) staat

- → ook: instance variable, instantievariabele, field, data member

`private int prijs;`	`private`	De access modifier (meestal `private`)
	`int`	Het datatype
	`prijs`	De naam v.h. attribuut

- elke instantie van een klasse heeft geheugenplaatsen voor de attributen van die klasse
- deze attributen kunnen we beschrijven/uitlezen

`mijnInstantie.voornaam = "Dieter";`	`mijnInstantie`	De naam van een variabele: - die van het type Persoon is én - waaraan een instantie van Persoon is toegekend
	`voornaam`	De naam van het attribuut waarnaar je een waarde wilt wegschrijven
	`"Dieter";`	De waarde die je wilt toewijzen

Instantiëren

NaamKlasse mijnInstantie = new NaamKlasse();	`NaamKlasse`	Een nieuw datatype dat automatisch werd gespecificeerd
	`mijnInstantie`	De naam v.d. variabele
	`new`	Keyword om een nieuwe instantie te creëren
	`NaamKlasse();`	De naam van de klasse waarvan we een nieuwe instantie willen creëren

→ instantiëren gebeurt nooit in de body van de klasse zelf, alleen op andere plaatsen

Getters (accessoren) en setters (mutatoren)

Wanneer we een *private* attribuut willen uitlezen vanuit code maken we gebruik van een **getter**

- getters zijn altijd public (doel = de waarde v.e. private attribuut publiek te maken)
- het return type is identiek aan het type van het overeenkomstige attribuut
- er zijn geen parameters
- de naam begint altijd met "get", gevolgd door de naam van de property die je wilt instellen
  - → bv. txtVoornaam.getText();
- getters die een waarde van het type boolean teruggeven, beginnen met de prefix "is"
  - → bv. txtVoornaam.isEnabled();
- wordt altijd in lowerCamelCase geschreven (hoofdlettergevoeligheid!)

Wanneer we de waarde van een *private* attribuut willen veranderen vanuit code gebruiken we een **setter**

- ook setters zijn altijd public (doel = de waarde v.e. private attribuut instelbaar te maken van buiten)
- het return type is altijd void
- er is slechts één parameter, van hetzelfde type als het overeenkomstige attribuut
- de naam begint altijd met "set", gevolgd door de naam van de property die je wilt instellen
  - → bv. txtVoornaam.setText();
- wordt altijd in lowerCamelCase geschreven (hoofdlettergevoeligheid!)

Automatisch genereren

Netbeans → Refactor > Encapsulate fields > Select all > Refactor

Voorbeeld

```
public class Persoon {
   private String voornaam;
   public String getVoornaam(){
      return this.voornaam;
   }
   public void setVoornaam(String nieuweVoornaam) {
      this.voornaam = nieuweVoornaam;
   }
}
```

Methoden

= een functie die gedefinieerd staat in een klasse (een specifiek geval van een functie)

This

= verwijzing in de klassedefinitie naar de instantie waarop de methode zal worden aangeroepen

- → hiermee kunnen we binnen een methode die we schrijven andere methoden/attributen van dat object aanroepen
- → bv.

```
public class Persoon {
   public String voornaam;
   public String achternaam;

   public String volledigeNaam() {
       return this.voornaam + " " + this.achternaam;;
   }
}
```

Constructoren

= iets wat je in de body van de klassedefinitie schrijft, en wat aangeroepen wordt bij het aanmaken van een nieuwe instantie van die klasse

- → bevat code die zal worden uitgevoerd op het moment dat je een nieuwe instantie v.d. klasse maakt (zal meestal code zijn om de attributen van het nieuwe object te initialiseren met een bep. waarde)
- → attributen krijgen een standaardwaarde als ze niet geïnitialiseerd worden door een constructor
- → er wordt een default constructor gegenereerd als er niet één expliciet wordt geschreven
  - ↳ wordt aangeroepen door het keyword "new", gevolgd door de naam van de klasse en haakjes (bv. `Persoon student1 = new Persoon();`)

```
public Persoon() {}
```

`public`	Access modifier (meestal public)
`Persoon`	De naam v.d. klasse waarvoor je een constructor schrijft
`()`	Eventueel parameters
`{}`	De body van de constructor

Overloading

= constructoren met dezelfde naam, maar met een verschillende signatuur (m.a.w. de volgorde van de parametertypes en/of het aantal parameters verschilt)

- → laat o.a. toe om bepaalde standaardwaarden te voorzien in bepaalde gevallen
- → Binnen een constructor een andere (overloaded) constructor van dezelfde klasse oproepen, doe je met "this"

Voorbeeld

```
public Persoon(String voornaam, String achternaam, String woonplaats) {
   this(voornaam, achternaam, 18);
   this.woonplaats = woonplaats;
}
```

Static en main

Static

= methoden en/of attributen gedefinieerd binnen een klassedefinitie, die gedeeld worden door alle instanties van die klasse

- → worden aangeduid met het keyword "static"
- → worden direct opgeroepen op de klasse (≠ een klasseinstantie)

Statische attributen

- → ook: class variable, klassevariabele, static field

```
private static int prijs;
```

`private`	De access modifier (meestal `private`)
`static`	Maakt het attribuut statisch
`int`	Het datatype
`prijs`	De naam v.h. attribuut

Statische methodes

- → ook: class variable, klassevariabele, static field
- → worden vaak gebruik om bibliotheken van functionaliteit in een klasse te verzamelen
- → binnen een statische methode kan je enkel statische attributen/methoden van de klasse waarbinnen je ze schrijft aanroepen

```
private static void doeIets(int c);
```

`private`	De access modifier (meestal `private`)
`static`	Maakt de methode statisch
`void`	Het return type
`doeIets`	De naam v.d. methode
`int c`	De parameters

Voorbeelden

Math

bevat een hele reeks (statische) methodes die wiskundige bewerkingen voorstellen

- → bv. `Math.cos(3.2);`

JOptionPane

bevat een hele reeks methodes die verschillende types van messageboxes op het scherm kunnen zetten

- → bv. `JOptionPane.showMessageDialog(null);`

public static void main()

= de eerste methode die wordt aangeroepen wanneer je je programma start

- → bij ons wordt deze methode vanzelf gegenereerd wanneer we een nieuwe JFrame Form aanmaken
- → er wordt dus een instantie gemaakt van de GUI-klasse die je zelf hebt gegenereerd
- → de instantie van de GUI klasse die wordt gemaakt bij het opstarten van het programma, blijft bestaan tot op het moment dat de applicatie wordt afgesloten.

Stringfuncties

Werkt door een string op te slaan in een variabele, de stringmethode op te roepen op deze variabele en eventueel het resultaat op te vangen in een andere of dezelfde variabele.

```
String richting = “Multec”;
richting = richting.replace(‘u’, ‘a’);
```

variabele.**charAt**(*index*)	Returnt het karakter op een bepaalde index
variabele.**contains**(*string*)	Returnt true als *variabele string* bevat
variabele.**endsWith**(*string*)	Returnt true als *variabele* eindigt met *string*
variabele.**equals**(*string*)	Returnt true als *variabele* hetzelfde is als *string*
variabele.**indexOf**(*string*[, *start*])	Returnt de index van *string* in *variabele*, eventueel vanaf *start*
variabele.**lastIndexOf**(*string*)	Returnt de laatste index van *string* in *variabele*
variabele.**length**()	Returnt de lengte (aantal karakters) van *variabele*
variabele.**matches**(*regex*)	Returnt true als *variabele* voldoet aan de reguliere expressie
variabele.**replace**(*old*, *new*)	Returnt een string volgens *variabele*, met alle instanties van *old* vervangen door *new*.
variabele.**split**(*regex*, *limiet*)	Returnt een array van gesplitste strings, op basis van *variabele* en volgens de reguliere expressie *regex*.
variabele.**startsWith**(*string*)	Returnt true als *variabele* start met *string*
variabele.**substring**(begin[, eind])	Returnt een substring van *variabele*, van de *begin*index tot (eventueel) een *eind*index
variabele.**toCharArray**()	Returnt een array van chars op basis van *variabele*
variabele.**toLowerCase**()	Zet *variabele* om naar kleine letters
variabele.**toUpperCase**()	Zet *variabele* om naar hoofdletters
variabele.**trim**()	Returnt een kopie van *variabele*, zonder eventuele spaties voor of na de string.

UML

UML

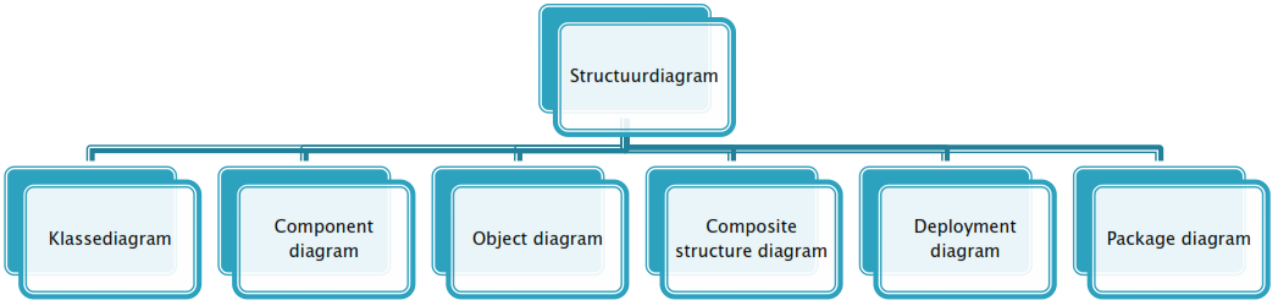
= Unified Modelling Language; een grote set van grafische notaties om visuele modellen op te bouwen om software te ontwerpen

- → niet alle soorten modellen worden gebruikt
- → wij bekijken het klassediagram

Modellen

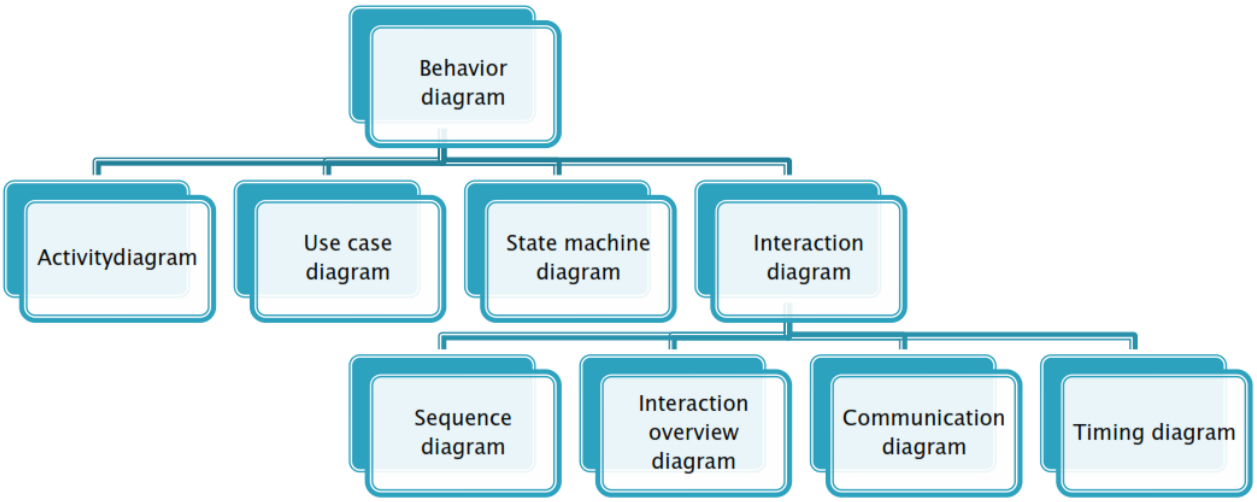
Statische modellen (structuurdiagram): visualiseren de statische structuur van een systeem

- → relaties en opsommingen van objecten, attributen, operaties.
- → bv. klassediagram



Dynamische modellen (behavior diagram): visualiseren het dynamische gedrag van het systeem

- → tonen het aanroepen van operaties binnen objecten, aanpassingen aan statussen, …



Klassendiagram

= statische relaties tussen componenten van een systeem modelleren

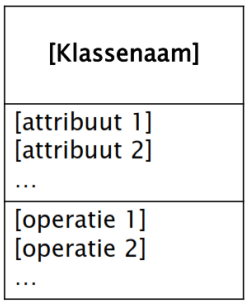
→ er kunnen meerdere klassendiagrammen bestaan die eenzelfde systeem vanuit verschillende standpunten beschrijft (bv. klassen met methodes, enkel de klassenamen)

Klasses

= vierkant met de naam van de klasse erin (zelfstandig naamwoord)

→ kunnen één of meerdere objecten worden wanneer de applicatie wordt uitgevoerd
→ voorstelling door compartimenten: klassenaam, attributen, operaties, exceptions, …
→ sommige attributen en operaties kunnen verborgen worden als ze niet relevant zijn voor het schema (aangeduid door elipsis: **…**)

Klasse met één compartiment => bevat klassenaam



Klasse met meerdere compartimenten

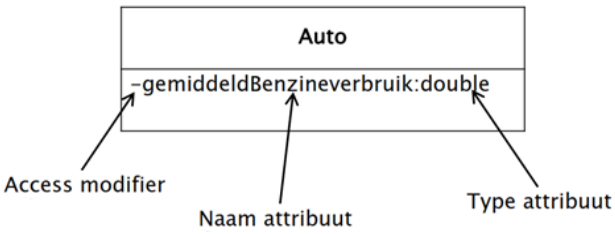
multipliciteit: duidt aan hoeveel instanties (objecten) er van de klasse kunnen bestaan d.m.v. een nummer rechtsboven in de klasse (standaard/ geen aanduiding: meerdere instanties mogelijk)

singleton: een klasse waarvan er maar één instantie kan worden gemaakt

Attributen

= de velden van de klasse

→ **2 voorstellingen:**
- ✓ door tekst in het vierkant dat de klasse voorstelt
- ✓ door een relatie tussen 2 klassen



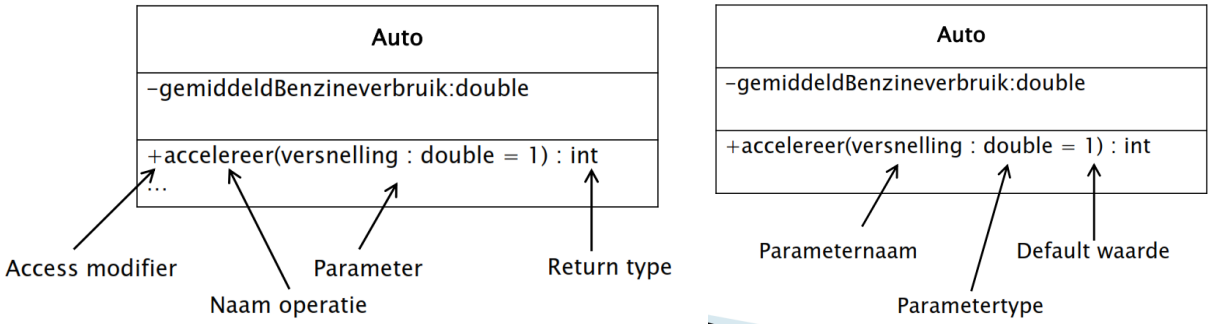
multipliciteit (optioneel): kan weergegeven worden tussen vierkante haakjes als je attribuut een collectie is (bv. [5]).

extra kenmerken (optioneel): kan weergegeven worden tussen accolades (bv. {unique, ordered})
- ✓ ordered (gesorteerd)
- ✓ unique (geen duplicaten toegelaten; standaard)
- ✓ not unique (wel duplicaten mogelijk, bv. namen van personen)
- ✓ readonly (attribuut mag niet meer aangepast worden nadat het werd ingesteld)

Operaties

Duiden aan wat een klasse kan doen (= methodes)

- → meestal in derde compartiment van klasse
- → indien er extra irrelevante operaties zijn, kunnen we dit ook met een ellipsis (…) aanduiden
- → meerdere parameters worden gescheiden door een komma



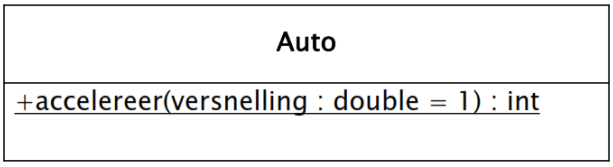
Getters en setters

zijn ook methoden van een klasse

- → komen overeen met een bepaald attribuut van die klasse
- → vaak worden zij niet beschreven in een klassediagram (zijn niet relevant genoeg om beschreven te worden in het UML-diagram)

Static

Statische attributen en operaties worden onderlijnd



Packages en namespaces

Java laat ons toe om de klasses die we aanmaken te ordenen in zogenaamde *packages*

- → laten toe om klasses in te delen in zogenaamde namespaces
- → laten toe om op een eenduidige manier te verwijzen naar een klasse in uw programma

Kenmerken

- ✓ In een package kunnen we één of meerdere klasses/klassedefinities aantreffen
- ✓ In een package kunnen we geen 2 klasses hebben met dezelfde naam
- ✓ In het *pad* dat gebruikt wordt om een klasse uniek te identificeren op uw computer wordt een punt gebruikt om de packages te onderscheiden
- ✓ wanneer je een klasse gebruikt, moet je in principe altijd de naam van de klasse en het pad naar de package aanduiden (behalve als de klasse in dezelfde package/namespace gedefinieerd is als de klasse waarvan je gebruik wilt maken)

Notatie

- → het "pad" naar de package van een klasse wordt aangeduid met punten (bv. com.ford.Auto, com.tesla.Auto)
- → de namen van packages schrijven we volledig met kleine letters (begint vaak met de omgekeerde URL van het bedrijf dat de klasse heeft geschreven)

Importeren

Je kan packages importeren, zodat je bij de implementatie van de klasse het pad van de package niet telkens moet vermelden

Voorbeeld

```
package be.heylen.maarten;
import com.tesla.Auto;          // De Auto-klasse van een package importeren
import com.tesla.*;             // Alle klasses van een package importeren
```

Eigen klasses in packages zetten

We kunnen op 2 momenten beslissen in welke package we een bepaalde klasse willen zetten
(je moet altijd op één van de twee manieren aanduiden waarbij de package hoort)

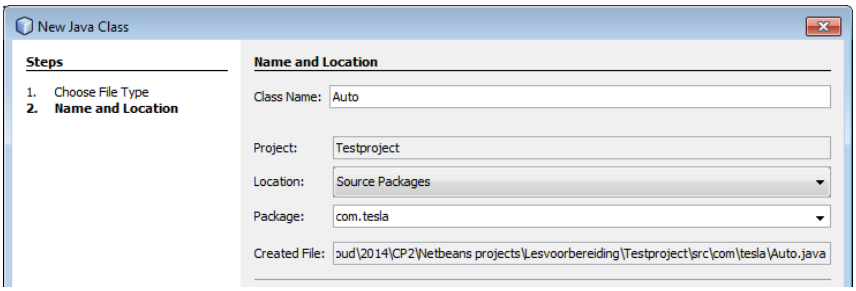
Het instellen van een package bij een bestaande klasse

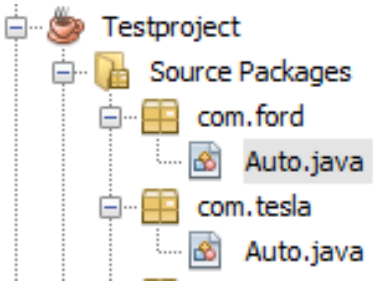
- → het keyword "package" + het pad van je package
- → buiten je klassedefinitie, maar binnen dezelfde file waarin je je klassedefinitie schrijft
- → de bestanden met de klassedefinities ordenen we in een folder met hetzelfde "pad" als de package (bv. `<projectfolder>\com\ford\Auto.java`)

Voorbeeld

```
package com.tesla;
public class Auto {
   //Uw attributen
   //Uw getters en setters
   //Uw constructoren
   //Uw methoden
}
```

Het instellen van een package bij een nieuwe klasse





De verschillende klasses (met eventueel dezelfde naam) in de verschillende packages worden ook gevisualiseerd in de project explorer van Netbeans

Begrippenlijst

Algemeen

IDE *(Integrated Development Environment)*: een (veredelde) teksteditor die gebruikt wordt om code te schrijven; bevat extra functionaliteiten om het ontwikkelen van programma's te vergemakkelijken

→ er bestaan verschillende 'merken', bv. Netbeans, Eclipse, Visual Studio, Processing IDE

programmeertaal: een codetaal waarmee je programma's kan schrijven die bepaalde dingen kunnen doen (bv. Java)

softwarebibliotheek: een hoop code die in een bestaande programmeertaal geschreven is en die je kan aansturen gebruik makende van diezelfde taal (bv. Processing)

syntax: de set van regels die je moet volgen om een bepaalde taal correct te schrijven (of te spreken)

objectgeoriënteerd programmeren: een zeer belangrijk concept, die je toelaat op een makkelijke manier code te hergebruiken, op een beheersbare manier complexere programma's te schrijven en meer gestructureerd te werken.

Swing: een veelgebruikte softwarebibliotheek voor Java die je toelaat om snel en eenvoudig een functionele GUI (Graphical User Interface) te maken voor je programma.

→ een **jFrame** stelt een top-level programmavenster voor die een titelbalk en een rand heeft en die dus de basis vormt van de programma's die wij zullen maken

Netbeans: de professionele IDE die wij zullen gebruiken om onze programma's te schrijven

camelCase: de eerste letter is een kleine letter; elk volgend woord wordt begonnen met een hoofdletter en alle woorden worden aan elkaar geschreven

PascalCase (of **UpperCamelCase**): de eerste letter is een hoofdletter; elk volgend woord wordt begonnen met een hoofdletter en alle woorden worden aan elkaar geschreven

event: een gebeurtenis die zich voordoet, en waar we op een bepaalde manier op kunnen reageren door middel van een programma/*event handler* (bv. drukken op een muisknop/toets)

member: verwijst naar ofwel een field of een methode van een klasse.

property: eigenschap; afhankelijk van de context verwijst dit naar de accessoren en mutatoren of de fields. Meestal wordt hiermee verwezen naar de accessoren en de mutatoren.

access modifier/specifier: bepaalt de mate waarin het attribuut/methode/… waar hij bijstaat toegankelijk is (bv. public, private, protected)

object: een instantie van een klasse (het resultaat van een instantiëring)

Variabelen

variabele: een plaats in het geheugen met een daaraan gekoppelde naam en datatype

instantievariabele: field/attribuut/data member

klassevariabele/static field: een variabele die wordt gedeeld door alle instanties van een klasse.

lokale variabele: een variabele die gekend is binnen een codeblok, maar die niet toegankelijk is van buiten dat codeblok.

scope van een variabele: het gebied binnen de code waarbinnen de variabele gekend is.

Functies en methoden

functie: een groep opdrachten die bij elkaar hoort en die een naam heeft

methode: een functie die gedefinieerd staat in een klasse (een specifiek geval van een functie)

instantiemethode: een methode die we enkel kunnen aanroepen op instanties van een klasse, en niet op de klasse zelf (een specifiek geval van een methode).

- → zullen in regel verschillende acties uitvoeren met de waarden die in de attributen opgeslagen zitten

klassemethode/statische methode: methode die direct wordt opgeroepen op de klasse, zonder dat er een instantie aangemaakt moet worden

argument: de effectieve waarde die je voor een parameter meegeeft bij het aanroepen van de methode

- → bv. `mijnFunctie("argument")`
- → net zoals alle andere variabelen moet je ook aan een parameter-variabele een waarde toekennen van hetzelfde type als dat in de declaratie van de parametervariabele staat

signatuur van een methode: de combinatie van de naam, de volgorde van de types van parameters en het aantal parameters

overloaded methodes/functies: methodes/functies met dezelfde naam, maar met een verschillende signatuur (maw: de volgorde van parametertypes en/of het aantal parameters verschilt)

terugkeertype/return type: het type van de waarde die de methode/functie zal teruggeven naar de plaats vanwaar hij wordt aangeroepen.

parameter: een lokale variabele die je een waarde kan geven bij het aanroepen van de methode/functie en die alleen gekend is binnen de methode/functie waar hij bijstaat. Parameters hebben een type en een naam.

Getters en setters

getter (accessor): een methode die je kan gebruiken om de waarde van een field uit te lezen.

setter (mutator): een methode die je kan gebruiken om de waarde van een field in te stellen.