Creative Programming III

Netbeans

Getters en setters genereren

Rechtsklik > Insert Code… > Getter and Setter…

Constructors genereren

Rechtsklik > Insert Code… > Constructor…

JAR-bestand toevoegen aan project

Project explorer > Libraries > Rechtsklik > Add JAR/Folder…

System.out.println

= methode die je toelaat om een string weg te schrijven naar de console

bv. `System.out.println("Dit is een test");`

Access modifiers

- **public**: member kan van binnen of buiten de klasse aangeroepen worden
- **private**: member kan enkel van binnenin de klasse aangeroepen worden
- **protected**: member kan van binnen de klasse + overervende klassen aangeroepen worden

Externe klassebibliotheken (= libraries)

= de code van een andere programmeur hergebruiken in je programma

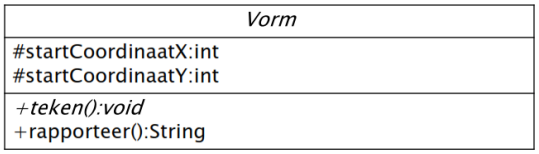
- één van de grote redenen waarom we objectgeorienteerd programmeren is het hergebruik van klasses
- bij Java → meestal d.m.v. JAR-bestanden (Java Archive File; een ZIP-bestand met andere extensie)
- JAR-bestanden bestaan uit
    - metadata, afbeeldingen, etc.
    - .class-bestanden (gecompileerde .java-bestanden; niet meer leesbaar, maar wel nog bruikbaar)

UML

Access modifiers

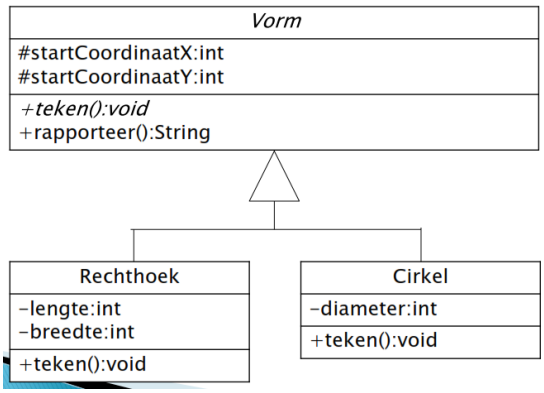
private	→	-
public	→	+
protected	→	#

Abstracte klasses/methodes

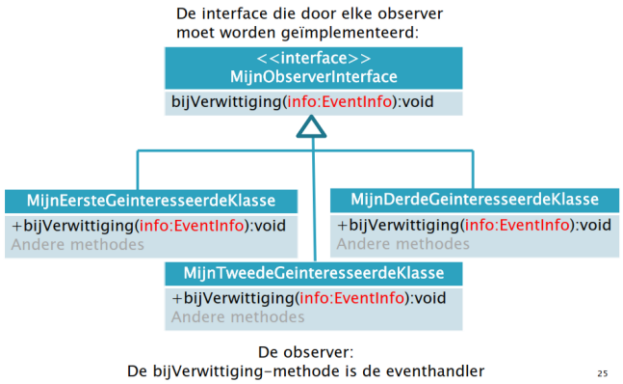


Abstracte klasses (overerving)

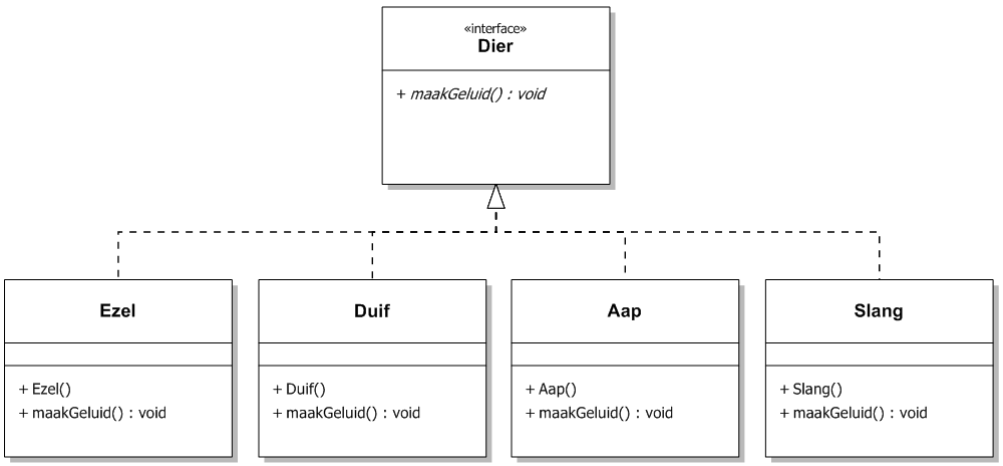
Rechthoek en cirkel erven over van extenden Vorm



Observer Pattern



Interfaces



Bestanden lezen/schrijven

javax.swing.JFileChooser

= Swing-klasse waarvan een instantie een keuzevenster voor bestanden voorstelt

Voorbeeld

```
JFileChooser mijnFileChooser = new JFileChooser();
File gekozenFile;

switch(mijnFileChooser.showOpenDialog(this)) {
   case APPROVE_OPTION:
       // Er is een bestand gekozen
       gekozenFile = mijnFileChooser.getSelectedFile();
       break;

   case ERROR_OPTION
       // Er is een bestand gekozen, maar een fout gebeurd
       break;

   case CANCEL_OPTION
       // Er is geannuleerd
       break;
}
```

java.io.File

= standaardklasse waarvan een instantie een bestand of directory op je schrijf voorstelt

- het pad wordt meegegeven met de constructor
  - **Windows**: `new File("C:\\pad\\naar\\mijn\\muziek\\track.mp3")`
  - **OS X**: `new File("/pad/naar/mijn/muziek/ track.mp3")`
- met een instantie kan je o.a.
  - bestandseigenschappen opvragen
  - navigeren t.o.v. het bestand
  - acties uitvoeren op de folder/het bestand

Het hele bestand in één keer lezen

= de static methode `readAllLines` uit de `java.nio.file.Files` klasse

Voorbeeld

```
File mijnBestand = new File("c:\pad\naar\bestand");
try {
   List<String> inhoud = Files.readAllLines(mijnBestand.toPath());
   if (inhoud.size() != 0) {
       for (String huidigeRegel : inhoud) {
          System.out.println(huidigeRegel);
       }
   }

} catch (IOException ex) {
   // Geef bijvoorbeeld een aangepaste foutboodschap
   JOptionPane.showMessageDialog(this, "Fout");
}
```

Het hele bestand in één keer schrijven

= de static methode `write` uit de `java.nio.file.Files` klasse

Voorbeeld

```
File mijnBestand = new File("c:\pad\naar\bestand");
try {
   ArrayList<String> inhoud = new ArrayList<String>();
   inhoud.add(txtEersteRegel.getText());
   inhoud.add(txtTweedeRegel.getText());
   inhoud.add(txtDerdeRegel.getText());
   Files.write(mijnBestand.toPath(), inhoud, StandardOpenOption.APPEND);

} catch (IOException ex) {
   //Geef bijvoorbeeld een aangepaste foutboodschap
   JOptionPane.showMessageDialog(this, "Fout");
}
```

Belangrijkste opties om het doelbestand te openen

Optie	Betekenis
APPEND	Het bestand wordt klaargemaakt om ernaar te schrijven. De tekst die moet worden geschreven wordt aan het einde van het bestand toegevoegd.
CREATE	Er wordt een nieuw bestand aangemaakt indien het bestand nog niet bestond
CREATE_NEW	Er wordt een nieuw bestand aangemaakt. Indien het bestand reeds bestond krijg je een fout.
DELETE_ON_CLOSE	Het bestand wordt verwijderd bij het afsluiten van de schrijfopdracht.
WRITE	Het bestand wordt klaargemaakt om ernaar te schrijven.

Het bestand regel per regel lezen

= de `BufferedReader` klasse uit de package `java.io`

Voorbeeld

```
File bestand = new File("c:\pad\naar\bestand");
try {
   BufferedReader mijnBR = Files.newBufferedReader(bestand.toPath());
   String uitgelezenRegel = mijnBR.readLine();
   while (uitgelezenRegel != null) {
       System.out.println(uitgelezenRegel);
       uitgelezenRegel = mijnBufferedReader.readLine();
   }

} catch (IOException ex) {
   // Geef bijvoorbeeld een aangepaste foutboodschap
   JOptionPane.showMessageDialog(this, "Fout");
}
```

Het bestand regel per regel schrijven

= de `BufferedWriter` klasse uit de package `java.io`

```
File bestand = new File("c:\pad\naar\bestand");
try {
   BufferedWriter mijnBW = Files.newBufferedWriter(bestand.toPath(),
   StandardOpenOption.APPEND);
   mijnBW.write(txtEersteRegel.getText());
   mijnBW.newLine();
   mijnBW.write(txtTweedeRegel.getText());
   mijnBW.close();

} catch (IOException ex) {
   // Geef bijvoorbeeld een aangepaste foutboodschap
   JOptionPane.showMessageDialog(this, "Fout");
}
```

Swing

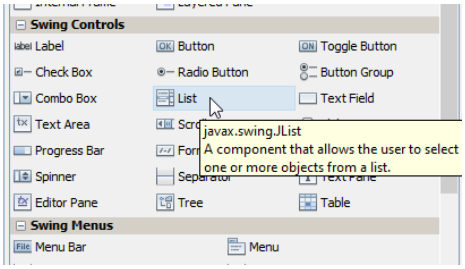
javax.swing.JTextArea

= Swing-klasse waarvan een instantie een textveld met meerdere lijnen voorstelt

javax.swing.JList

= Swing-klasse waarvan een instantie een lijst voorstelt waaruit je één of meerdere items kan selecteren

1. Voeg een JList toe in de designer van Netbeans



2. Voeg een DefaultListModel() instantie toe aan de model-property van je JList

```
// Mits import
import javax.swing.DefaultListModel;

// In de constructor, na het aanroepen van initComponents()
DefaultListModel model = new DefaultListModel();
OF
DefaultListModel<Persoon> model = (DefaultListModel<Persoon>) lstTest.getModel();

lstLijst.setModel(model);
```

3. Items toevoegen/verwijderen

(eender waar na het initialiseren van het model)

```
Persoon item = new Persoon("Jan", "Peeters", 45);
mijnModel.addElement(item);
mijnModel.removeElement(item);
```

4. Eventueel de toString methode v. je objectklasse overriden om weergave i.d. JList aan te passen

```
// In de Persoon-klasse
public String toString() {
   return this.voornaam + " " + this.achternaam;
}
```

Interessante properties van de JList

- **selectedIndex** Geeft de index van het item dat op dat moment geselecteerd is
- **selectedIndices** Geeft een array terug van de indices die op dat moment geselecteerd zijn
- **selectionMode** Bepalen of de gebruiker al dan niet versch. items tegelijkertijd kan selecteren
- **getSelectedValue** Geeft een referentie naar het eerste geselecteerde object
- **getSelectedValuesList** Geeft een lijst terug van alle objecten die op dat moment geselecteerd zijn

Interessante events van de JList

- **ValueChanged** Gefired wanneer er een verandering is in het aantal/de geselecteerde items

Escape sequences

Tekens die binnen Java een speciale betekenis hebben moeten binnen een string ge-escaped worden met een backslash \

- dit concept komt ook in andere programmeertalen terug
- de volgende acties zijn nodig om naar een volgende regel over te gaan in een tekstbestand:
  - ✓ op Windows-systemen: \r\n (2 karakters achter elkaar)
  - ✓ op Linux en OS X-systemen: \n
  - ✓ op (heel oude) Mac-systemen: \r
  - ✓ voor andere types van computers: andere combinaties
- bv. `"Ik hou van \"Java\""`

Escape sequence	Overeenkomstig karakter
\\	Backslash
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline
\r	Carriage return
\f	Formfeed
\b	Backspace

Enumeration

= type variabele, zoals integer, string, etc.

- symboliseert een bepaalde status (bv. de windrichtingen, de seizoenen, de weekdagen)
- bevat altijd een beperkt en vast aantal waarden
- maakt je code minder foutgevoelig en is korter dan een reeks aparte variabelen te maken

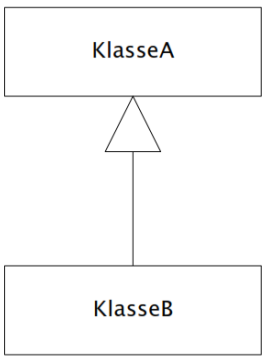
Voorbeeld

```
public enum Richting { NOORD, OOST, ZUID, WEST }
Richting richting = Windrichting.NOORD

switch(richting) {
   case NOORD:
       gaVooruit();
       break;

   case WEST:
       gaNaarRechts();
       break;
}
```

Overerving

Klasse A erft over van klasse B

- → klasse B krijgt alle attributen/methoden van klasse A
- → klasse A is superklasse / moederklasse / base class / parent class
- → klasse B is child class / subclass / derived class / extended class



- een klasse kan slechts van één andere klasse overerven
- meerdere klasses kunnen van één moederklasse overerven
- doel = hergebruik van code
- wordt aangeduid met het woord **extends**

```
public class KlasseB extends KlasseA
public class Student extends Persoon
```

Abstractie

Abstracte klasse

= klasse waar je geen instantie van kan maken

- bevat:
  - abstracte methoden (zonder implementatie)
    bv. `public abstract void doeIets();`
  - gewone methoden (met implementatie)
    bv. `public abstract void doeIets() {};`
  - attributen en constructoren
    bv. `String naam;`

- de overervende klasse gebruikt het woord *extends*
  bv. `public class Rechthoek extends Vorm`

Interface

= een contract waaraan andere klassen zich moeten houden

→ is een leeg omhulsel → kan geen implementaties van methoden bevatten, enkel signaturen
→ klassen die een interface implementeren moeten alle methoden in die interface te implementeren
→ verschillen met abstracte klasses:
  - abstracte klasses kunnen geimplementeerde methoden hebben, interfaces niet
  - een klasse kan meerdere interfaces *implementeren*, maar kan maar één abstracte klasse *extenden*
  - een klasse kan tegelijk een klasse *extenden* en van andere klasses *implementere*
→ waarom werken met interfaces? **polymorfismen** gebruiken
  (variabele van een algemeen type koppelen aan een instantie van een specifieker type)

Voorbeeld

```
public interface IDier {
   public void maakGeluid();
   public void stappen();
}

public class Duif implements IDier {
   public void maakGeluid() {
       System.out.println("roekoe roekoe");
   }

   public void stappen() {
       System.out.println("roekoe roekoe");
   }

   // Eventuele extra methoden die niet door de interface gespecifieerd zijn
}
```

Super

= keyword waarmee je vanuit de constructor v.e. subklasse een constructor v.e. superklasse kan aanroepen

- moet de eerste lijn code zijn in de body van een constructor
- een constructor in een subklasse moet steeds een constructor van de superklasse aanroepen

Object-klasse

= de klasse waar elke in Java gekende klasse van overerft (alle klassen zijn een subklasse van Object)

- omvat een aantal methoden die dus overal terugkomen (o.a. `toString`, `equals`, `getClass`)
- wordt meestal gebruikt met polymorfisme
  - ✓ wanneer een methode als argument een instantie van eender welke klasse accepteert
  - ✓ wanneer een methode als return value een instantie van eender welke klasse teruggeeft
  - ✓ wanneer je een lijst van objecten van eender welke klasse wilt bijhouden

  ↳ deze waarden moeten dan wel nog gecast worden naar hun eigenlijke klasse
  Voorbeeld:

```
List<Object> lijst = new ArrayList<>();
lijst.add(new Persoon());
Persoon ikke = (Persoon) lijst.get(0);
```

toString()

= geeft een tekstuele representatie van een object

- **standaard**: klassenaam + @ + unieke identificatie van de instantie (bv. `WC2_1.Persoon@6d06d69c`)
- zullen we meestal **overriden** om een meer betekenisvolle string terug te krijgen

Overriding

= biedt de mogelijkheid om een methode van de superklasse te overschrijven om een specifiekere implementatie te kunnen bieden

Voorbeeld

```
@Override
   public void toString() {
   return voornaam + " " + achternaam;
}
```

Polymorfisme

= een instantie van een child class B opslaan in een variabele van het type van een bepaalde superklasse A

- bv. `Persoon eenPersoon = new Student("Jan", "Peeters", 18, 1);`
- enkel de methoden van de superklasse Persoon zijn hier toegankelijk, want het type is Persoon
- als we toch toegang willen tot de methodes en fields van het Student-object, dan:
  - moeten we eerst zeker zijn dat de variabele wel degelijk verwijst naar een object van het type Student, d.m.v. het keyword *instanceof*
  - moeten we het object casten naar een variabele van het type Student
  - pas dan kunnen we de specifieke methodes en fields gebruiken

```
if (ehbPersoon instanceof Student) {
   Student deStudent = (Student) ehbPersoon;
   int leerjaar = deStudent.getLeerjaar();
   int leeftijd = deStudent.getLeeftijd();
}
```

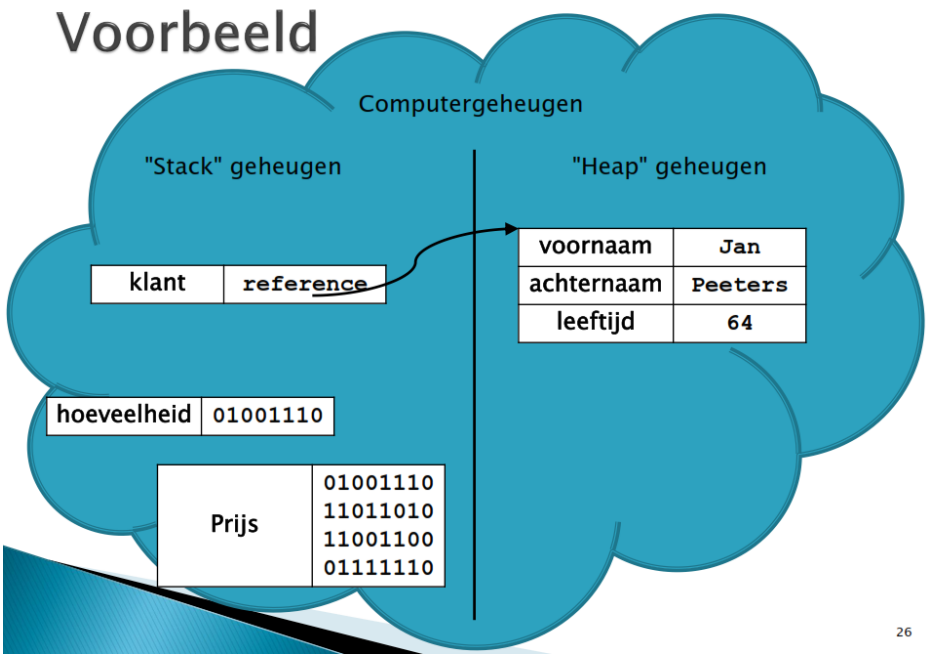
- wordt vooral gebruikt
    - wanneer een methode als argument een instantie van verschillende soorten (child)klasses moet kunnen accepteren. We kiezen een superklasse als type voor onze parametervariabele.
    - wanneer een methode als return value een instantie van verschillende soorten (child)klasses moet kunnen teruggeven. We kiezen een superklasse als type voor de return type.
    - wanneer je een lijst van objecten van eender welke klasse wilt bijhouden

Primitive types en reference types

- **primitive datatypes** (byte, short, int, long, float, double, boolean, char)
    - technische specificaties liggen volledig vast in Java
    - hebben vooraf gedefinieerde standaardgroottes
    - worden opgeslagen in het stackgeheugen van de computer
    - standaardwaarde: hangt af van datatype (0 / 0.0 / false / …)

- **reference types** (klassen, arrays, etc.)
    - zijn samengesteld uit andere klasses, arrays of primitive types
    - hebben geen vaste standaardgrootte
    - worden door het memory management opgeslagen in het heapgeheugen van de computer
    - een referentie naar de plaats in het heapgeheugen wordt bijgehouden in het stackgeheugen
    - standaardwaarde: null

- **immutable** (o.a. String)
    - zijn eigenlijk objecten
    - na ze zijn aangemaakt, kan hun originele waarde niet meer veranderen (de originele waarde wordt verwijderd en een nieuwe waarde wordt gekoppeld aan de variabele)



Variabelen

Declareren van variabelen

- variabelen die binnen een functie gedeclareerd zijn en nog niet geïnitialiseerd zijn kunnen niet gebruikt worden
- variabelen die als field gedeclareerd zijn en nog niet geïnitialiseerd zijn kunnen wél gebruikt worden, want zij hebben een default value overeenkomstig hun type
- het niet initialiseren van variabelen geeft dikwijls aanleiding tot onverwachte situaties
    - primitieve datatypes → meestal een functionele fout (je krijgt niet het verwachte resultaat)
    - reference types → meestal een NullPointerException

Wijzigen van variabelen

- primitive datatypes: de waarde wordt rechtstreeks aangepast
- reference types:
    - ✓ een nieuw object wordt aangemaakt op de heap
    - ✓ de referentie wordt aangepast
    - ✓ de garbage collector zal het oude object automatisch uit het geheugen verwijderen
    - ↳ alleen bij het wijzigen van het hele object (≠ individuele fieldwaarden)
    - ↳ ook bij het toewijzen van null aan een reference type zal het originele object verwijderd worden

Kopiëren van variabelen

- primitive datatypes: er wordt effectief een kopie gemaakt van de data
- reference types: enkel de referentie wordt gekopieerd (er wordt dus geen nieuw object gemaakt)

Variabelen als argument

- altijd *by value* (er is een kopie van de oorspronkelijke variabele beschikbaar in de functie/methode)
- primitive datatypes: er wordt effectief een kopie gemaakt van de data
- reference types: enkel de referentie wordt doorgegeven (er wordt dus geen nieuw object gemaakt)

Variabelen vergelijken

Wanneer je 2 variabelen met elkaar vergelijkt d.m.v. de == operator, vergelijk je hun waarden in de stack

- primitive datatypes: de effectieve waarden worden vergeleken
- reference types: de referenties worden vergeleken
    - → als er op twee geheugenlocaties per toeval een object staat met dezelfde inhoud, zijn deze nog niet gelijk, omdat de referenties verschillend zijn)
    - → om na te gaan of twee objecten dezelfde inhoud hebben, gebruiken (en eventueel overriden) we de equals() methode

Wrapper-classes

In Java bestaat er van elk primitief datatype ook een klassevariant (een wrapper class)

- → deze klassen bewaren even goed een primitive type value
- → het verschil met de echte primitieve types is dat deze klasses extra functionaliteit aanbieden in de vorm van o.a. hulpmethodes
- → De naamgeving van de wrapper-classes verschilt in de meeste gevallen slechts met een hoofdletter van de naam van het primitief datatype

Collecties

Array

= een container die een vast aantal waarden van een voorafbepaald type kan bijhouden

- de lengte wordt vastgelegd wanneer de array gedefinieerd wordt
- elk item in een array wordt een **element** genoemd
- elk element in de array heeft een **index** (nummer waarmee naar dat element verwezen kan worden)
  - → na initialisatie kunnen hiermee waarden weggeschreven/opgevraagd worden
  - → bv. `mijnLijst[3] = 8;` (4de item, met index 3 en toegewezen waarde 8)
- een array is een **reference type**
- de lengte van een array kan opgevraagd worden door de fieldvariabele length
  - → bv. `System.out.println(mijnLijst.length);`

Declaratie (naar stackgeheugen)

```
int[] mijnLijst;
```

- **int** = het type van de (waarden binnen de) array
- **[]** = een vast merkteken om aan te duiden dat je een arraytype wilt declareren
- **mijnLijst** = de variabelenaam

Initialisatie (naar heapgeheugen)

```
mijnLijst = new int[5];
```

- **mijnLijst** = de variabelenaam
- **new** = de array is een reference type, we maken een nieuwe array aan
- **int** = het type van de (waarden binnen de) array (moet zelfde zijn als bij declaratie)
- **[5]** = merkteken van arrays + de lengte van de array

Declaratie en initialisatie met vooraf bepaalde waarden

```
int[] mijnLijst = {2, 1, 4};
```

- **int** = het type van de (waarden binnen de) array
- **[]** = een vast merkteken om aan te duiden dat je een arraytype wilt declareren
- **mijnLijst** = de variabelenaam
- **{2, 1, 4}** = de op te slagen waarden (het aantal waarden bepaalt de vaste lengte van de array)

Doorheen een array itereren

- **for-loop**

```
for (int i=0; i < array.length; i++) {
    System.out.println("Element: " + array[i]);
}
```

- **foreach-loop** (enhanced for loop)

```
for (String element : array) {
    System.out.println("Element: " + element);
}
```

Arrays kopieren

→ de System-klasse heeft een statische *arraycopy* methode, waarmee je eenvoudig arrays kan kopieren

Arraylist

= een container die een variabel aantal waarden van een bepaald type kan bijhouden

- er hoeft geen lengte vastgelegd te worden wanneer de arraylist gedefinieerd wordt
- houdt standaard elementen bij van het type Object, tenzij anders gespecificeerd (generics)
- primitive types worden automatisch omgezet naar hun overeenkomstige wrapper-class (autoboxing)

Vergeet niet te importeren

```
import java.util.ArrayList;
```

Een nieuwe ArrayList instantie aanmaken

```
ArrayList mijnLijst = new ArrayList();
```

Elementen toevoegen

```
mijnLijst.add("tekst");         // Voegt "tekst" toe op het einde van de array
mijnLijst.add(3, "tekst");      // Voegt "tekst toe op index 3, en schuift oude index
                                   3 en volgende waarden eentje op naar rechts
mijnLijst.set(3, "tekst");      // Vervangt de waarde op index 3 met "tekst"
mijnlijst.get(3);               // Haalt de waarde op index 3 op
```

Itereren

Zie *Doorheen een array itereren*

Boxing/unboxing

- **boxing**: het omzetten v.e. primitief datatype naar een instantie v. de overeenkomstige wrapper class
- **unboxing**: het omzetten v.e. instantie v.e. wrapper class naar het overeenkomstig primitief datatype

Generics

= laten toe om bij de declaratie/initialisatie van een klasse aan te duiden dat deze klasse alleen voor een bepaald type moet werken.

- aangeduid met <>
- bv. `Arraylist<Persoon> lijst = new Arraylist<Persoon>();`
  - → een arraylist die alleen kan werken met Persoonsobjecten.
  - → aangezien dat de arraylist weet dat er alleen maar Personen in de lijst kunnen zitten, moet je niet meer casten

Exception handling

Fouten (herhaling)

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

Exceptions

Stack trace

= een hiërarchisch spoor van alle methodes die werden aangeroepen om de thread op de huidige positie te krijgen, gebruikt bij het debuggen.

- → in het rood geprint in de console van Netbeans

Veelvoorkomende types

- **NullPointerException**
- **NumberFormatException**
  (doet zich voor wanneer je een Stringwaarde probeert om te zetten naar een getalwaarde (door middel van bv. `Double.parseDouble()`), maar de Stringwaarde niet de correcte vorm heeft)
- **ArithmeticException**
  (doet zich voor wanneer je in je programma wiskundige berekeningen uitvoert waarvoor de computer geen uitkomst kan berekenen)
- **ArrayIndexOutOfBoundsException**
  (doet zich voor wanneer je toegang probeert te krijgen tot een element van een array met een index die niet bestaat, bv. omdat de index groter is dan de grootte van de array)
- **IOException**
  (doet zich voor wanneer er iets fout gaat bij het lezen/schrijven van bestanden)

Exception handling

- in de eerste plaats het voorvallen van exceptions proberen te **vermijden**
  - ✓ code schrijven die de invoer van de gebruiker valideert
  - ✓ gebruik maken van hulpklasses/hulpcontrols die de gebruiker ondersteunen in het geven van de correcte invoer (bv. JFileChooser)

- daarnaast ook exception handling voorzien d.m.v. een **try-catch-blok**
  - → wanneer er zich een exception voordoet binnen de code van het try-block wordt
    - ✓ een instantie van de (subklasse van de) Exception-klasse aangemaakt en doorgegeven aan het catch-blok
    - ✓ de code in het catch block uitgevoerd
  - → als er zich geen exception voordoet
    - ✓ wordt het try-block volledig uitgevoerd
    - ✓ wordt het catch-block niet uitgevoerd

  - → wanneer er een exception optreedt, zeggen we dat er een exception wordt **gethrowd**
  - → wanneer er een een exception wordt afgehandeld in een catch-block, zeggen we dat de exception wordt **gecatched**

Voorbeeld

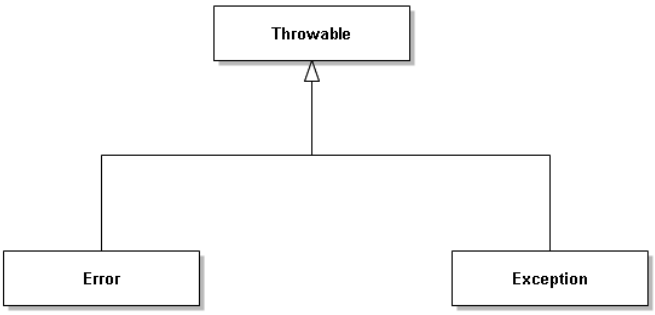
```
try {
   // Code die een runtime fout kan veroorzaken

} catch (Exception ex) {
   /* Code die alleen wordt uitgevoerd wanneer
   er zich een runtime fout heeft voorgedaan
   in het try-block. Voorbeelden: */

   System.out.println(ex.getMessage());
   ex.printStackTrace();
}
```

- er kan ook exception handling voorzien worden d.m.v. een **throws statement**
  - → extra documentatie om de persoon die de methode aanroept te laten weten dat deze methode een bepaald type exception kan throwen
  - → ervoor zorgen dat je een methode die een checked exception kan genereren niet in een try-catch block moet zetten in de methode waarin je ze aanroept

  - → bv. `void methodenaam() throws <exceptiontype>`

Exception types

Throwable



- Klassen die hiervan overerven
  - → kan je throwen
  - → kunnen gebruikt worden als argument van catch
- Bevat
  - → Execution stack
  - → Message die meer informatie geeft
- 2 grote types
  - ✓ Checked exceptions
    - ↳ exceptions die niet overerven van RuntimeException
    - ↳ moeten gecatched worden (d.m.v. try-catch-blok of throws statement)
  - ✓ Unchecked exceptions
    - ↳ exceptions die overerven van RuntimeException
    - ↳ duiden meestal op programeerfouten
    - ↳ moeten niet gecatched worden
    - ↳ bv. ArithmeticException, NegativeArraySizeException, NullPointerException

Errors

- worden gethrowd bij grote problemen die een normale applicatie niet moet catchen
- worden gegenereerd door Java
- komen zelden voor
- bv. VirtualMachineError, CoderMalfunctionError

Zelf exceptions throwen

De standaard exception classes gebruiken

1. Een instantie aanmaken van een bestaande exception klasse
2. Het keyword "throw", gevolgd door de instantie van de exception die je hebt gemaakt.

Voorbeeld

```
Exception mijnException = new Exception("dit is een test exception");
throw mijnException;
```

Zelf een exception type definieren

Waarom?

Extra informatie geven over de oorzaak en/of omstandigheden van de fout d.m.v. een eigen naamgeving en het toevoegen van extra attributen

Hoe?

een klasse maken die overerft van de klasse Exception of een klasse die overerft van Exception

Databases

Wat?

de data van onze applicatie persistent maken

→ m.a.w. ervoor zorgen dat data beschikbaar blijft wanneer onze applicatie wordt afgesloten

Hoe?

meestal d.m.v. een database

→ vanuit onze Java-applicatie ⟶ met behulp van JDBC (Java Database Connectivity Driver)
  ↳ API (application programming interface) om vanuit Java databases mee aan te sturen
  ↳ gemaakt voor relationele databases zoals MySQL, Oracle en MSSQL

Gebruikte klassen

de meesten zijn terug te vinden in de package `java.sql`, deze kan je best in één keer helemaal importeren

- **DriverManager**: stelt de driver voor die we gedownload hebben voor de specifieke database waarmee we willen werken
  → bv. `Connection connectie = DriverManager.getConnection(server, user, pass);`

- **Connection**: stelt een opgebouwde connectie met de database voor, waarnaar we uit te voeren opdrachten kunnen sturen
  → `prepareStatement(sqlQuery:String)`
  (geeft een PreparedStatement-object terug waaraan we nog parameters kunnen toevoegen)
  → `close()`
  (sluit de verbinding met de database, zodat resources terug vrijkomen)

- **PreparedStatement**: stelt een (mogelijks) geparametriseerde query naar de database voor
  → kan worden uitgevoerd met de executeQuery() of executeUpdate() methodes
    ✓ executeQuery() geeft resultaten terug
    ✓ executeUpdate() geef aantal aangepaste rijen terug
  → parameters in de query worden vervangen door ? en meegegeven door de methode `setObject(<index van ?>, <value>);`

- **ResultSet**: bevat de resultaten van jouw query
  → heeft een "pointer" die naar een bepaalde rij in de tabel verwijst
  → resultaten kunnen opgehaald worden met getters
  → de pointer kan verplaatst worden met methodes als `next()`, `first()`, etc.
  → itereren kan met een while-loop

```
while (mijnResultset.next()){
   String voornaam = mijnResultset.getString("Voornaam");
}
```

- **SQLException**: kan gethrowd worden door verschillende methoden in de SQL JDBC bibliotheek
  → moet verplicht opgevangen worden

Applicatiestructuur

Model

per tabel in de database een model-klasse in Java

- geef deze zo'n klasse evenveel attributen als de overeenkomstige tabel in de DB
- geef deze attributen een datatype dat overeen komt met de overeenkomstige kolom in de DB
- geef deze attributen dezelfde naam als de kolommen in de DB
- deze modellen worden verzameld in een package waarvan de naam eindigt met *model*

DAO's (Data Access Objects)

klasses die je toelaten om data met een database/datastore uit te wisselen

- stellen de toegangspoort naar onze database voorstellen in onze applicatie voor
- schermt de complexiteit van de databasetoegang af voor andere klasses in je applicatie
- bieden meestal methoden aan die de CRUD-acties implementeren (Create/Read/Update/Delete)
- moeten `static` zijn
- elke DAO implementeert methodes om één tabel in de database te manipuleren
- deze DAO's worden verzameld in een package waarvan de naam eindigt met *dao*

Database-klasse

- bevat de basisfunctionaliteiten van de DAO-klassen
- bevat als enige de connectiegegevens voor de database
  → is het enige dat aan deze klasse aangepast moet worden
- wordt mee opgenomen in het dao-package van je project

Swing

GUI-klasses

- deze klasses worden verzameld in een package waarvan de naam eindigt met *swing*

Webservices en Maven

Webservice

= een manier waarmee 2 computers met een gemeenschappelijke taal kunnen communiceren over een computernetwerk

- laat u toe om vanaf een client methodes aan te roepen op een servermachine
- er kan hierbij datauitwisseling plaatsvinden (argumenten meegeven / return values terugkrijgen)
- gebruikt meestal HTTP als onderliggend communicatieprotocol
  - → werkt volgens een client-servermodel
    - ✓ **servers** bieden diensten/services aan op het computernetwerk
      - ↳ is de service provider
      - ↳ kan verwijzen naar de computer waar het serverprogramma op loopt of naar het serverprogramma zelf
    - ✓ **clients** consumeren de diensten/services van de servers op het netwerk
      - ↳ is de service requester
      - ↳ kan verwijzen naar de computer waar het clientprogramma op loopt of naar het clientprogramma zelf
  - → werkt volgens een request-response patroon
    - ✓ de client stuurt een **request** (= aanvraag) naar de server om een service te gebruiken
    - ✓ de server voert een actie uit en stuurt een **response** (= antwoord) terug naar de client
- **data** (requests/responses) die wordt uitgewisseld via de webservice kan volgens verschillende vormen gestructureerd zijn (o.a. XML, JSON)
  - → wij gebruiken JSON (JavaScript Object Notation)
    - ✓ gemakkelijk leesbare manier om key-value pairs verzenden
    - ✓ vergelijkbaar met XML
    - ✓ internet media type = application/json
    - ✓ Voorbeeld

```
{"voornaam": "Maarten",
"achternaam": "Heylen"}
```

Soorten

er bestaan verschillende soorten webservices en op veel van deze soorten bestaan er ook nog varianten

- wij gebruiken **REST** (Representional State Transfer) webservices
  - ✓ één van de eenvoudigste soorten webservice
  - ✓ populair op dit moment
  - ✓ een REST webservice kan op verschillende manieren geïmplementeerd worden.
  - ✓ wij gebruiken een veel voorkomende variant die werkt op basis van HTTP GET/POST requests en JSON responses
  - ✓ wij maken REST webservices d.m.v. het Java Spring Framework

HTTP Requests

1. De client stuurt een HTTP request naar de server om de webservice aan te spreken
2. Op de server wordt een methode uitgevoerd die gekoppeld is aan die specifieke request
3. De server stuurt een HTTP response terug naar de client (bv. een string of JSON object)

GET

een querystring met een aantal naam/waardeparen wordt toegevoegd aan de URL

1. de querystring begint met een **?**
2. de keys en values worden gescheiden door een **=**
3. key/value pairs worden gescheiden door een **&**

Voordelen

- eenvoudig op te stellen vanuit de browser
- zijn zichtbaar in het adresveld en de geschiedenis van browser
- kunnen mee met een bookmark worden opgeslagen
- worden dikwijls op de server gelogd

Nadelen

- elke URL heeft een maximum lengte (soms max. 255 tekens)
- niet voor alle types gegevens
- zijn voor iedereen zichtbaar

Voorbeeld

```
http://www.bibliotheek.be/Auteur/getByName?naam="Shakespeare"
```

POST

Voordelen

- mogelijk om veel meer gegevens door te sturen.

Nadelen

- de doorgestuurde informatie is niet onmiddellijk zichtbaar
- de doorgestuurde informatie komt niet terug in bookmarks, history, … van browser

Voorbeeld

```
POST /studenten/registreer HTTP/1.1
Host: localhost
Connection: close
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac
OS X 10_6_4; de-de) AppleWebKit/533.16 (KHTML,
like Gecko) Version/5.0 Safari/533.16
Accept:
text/xml,text/html,text/plain,image/png,image/jpe
g,image/gif
Accept-Charset: ISO-8859-1,utf-8
Voornaam=Tim&Achternaam=Maes
```

AJAX

= Asynchronous Javascript en XML; een manier om op een asynchrone manier webrequests te lanceren vanuit Javascript

- zorgt ervoor dat je applicatie niet blijft hangen terwijl er een HTTP request wordt uitgevoerd
- gebruikt om REST webservices aan te spreken, (delen van) HTML-pagina's ophalen, etc.

Tools

- **Java**

- **Spring Framework**: een populair, open source applicatieontwikkelingsframework
  - ↳ bevat modules o.a. voor het maken van webservices, server scripted websites, authentication/authorization, dependency injection, etc.
  - ↳ maakt gebruik van een build automation tool
    (een stuk software dat een reeks taken die moeten worden uitgevoerd bij het compilen en starten van een nieuw gecodeerde applicatie automatiseert)
    - → zit ingebouwd in Netbeans (heel eenvoudig en beperkt)
    - → wij gebruiken Maven (nog voorbeelden: Gradle, Ant)

- **Tomcat webserver**: een http webserver die je webservice aanbiedt op het netwerk (via poort 8080)

- **Maven**: geavanceerde build automator die zorgt voor
  - ✓ het builden/runnen/compilen van je programma
  - ✓ het starten van de Tomcat webserver
  - ✓ het automatisch deployen van de webservice code naar de webserver
  - ✓ het onderhouden van dependencies, etc.

- **Tool om HTTP requests te genereren**
  - ✓ Webbrowser (alleen GET requests)
  - ✓ HttpRequester (een programma waarmee je meer mogelijkheden hebt om HTTP requests te versturen naar een server, zowel GET als POST requests)
  - ✓ website met AJAX
  - ✓ Android-app
  - ✓ etc.

Maven

- elk Mavenproject op deze aardbol moet een unieke **identificatiemogelijkheid** hebben die bestaat uit
  - ✓ een naam (Artifact ID) → meestal is zelfde naam als project goed
  - ✓ een groep (Group ID) → meestal is zelfde naam als basispackage/namespace goed
  - ✓ een versienummer (Version) → meestal is de standaardwaarde goed

- elk Mavenproject maakt een aantal **projectbestanden** aan
  - ✓ een Dependencies-folder
  - ✓ een pom.xml bestand met alle projectinstellingen en dependencies

Controllers

= de definitie van welke methodes worden gepubliceerd als webservice en wat deze methodes moeten doen

- meestal één controller per databasetabel en klasse van je model (om functionaliteit te groeperen)
- bij de naam van de klasse wordt een annotation gezet die aanduidt dat deze klasse een controller is
- bij de naam van de methode wordt een annotation gezet die aanduidt dat deze methode naar een bepaald pad op de webserver moet worden gepubliceerd
- meer info: bekijk de RestServerBasics en RestServerMetDatabase voorbeeldapplicaties

Events en observer pattern

Events

= een softwarematig bericht dat aanduidt dat er iets is gebeurd

- wordt uitgestuurd (*gefired*) door één bepaald object
- meerdere objecten kunnen zich inschrijven voor een bepaald event, waarmee zij aangeven dat zij op de hoogte willen worden gebracht wanneer het event zich voordoet.
- ingeschreven objecten reageren op het event door een stukje code (*event handler*) uit te voeren
- events vloeien voort uit het *observer design pattern*
- o.a. Swingcomponenten werken met events

Design Patterns

= oplossingen voor problemen die in het verleden reeds werden tegengekomen bij het ontwerpen van programma's

- in het kort: voorbeeldoplossingen voor veelvoorkomende problemen
- terug te vinden in de literatuur (bekendste boek: 'Design Patterns: Elements of Reusable object-oriented software' geschreven door de 'Gang of Four')
- NIET: kant-en-klare klassen
- WEL: een manier om je klassen te structureren om tot een goede oplossing te komen

Observer Pattern omvat 2 soorten klasses

- **observer** (de objecten die op de hoogte gehouden worden van de veranderingen van de subject)
  - → moet maar 1 methode hebben: de event handler
  - → deze methode moet een bepaalde signatuur hebben, afgedwongen met een interface
- **subject** (het object dat een verandering ondergaat en de observers daarvan op de hoogte zal brengen)
  - → houdt een lijst bij van alle observers (lijst van interface types)
  - → overloopt bij het voorkomen van het event de lijst van observers en roept de methode aan die volgens de interface op de observer geïmplementeerd moet zijn

Anonieme klasses

= klasse die geen naam heeft, maar die een bepaalde interface implementeert (verkorte schrijfwijze)

- laten u toe om in één keer een klasse te definiëren én te instantiëren.
- komt vaak voor ten behoeve van het afhandelen van events

Gewone klasse

```
public MijnListener implements
ActionListener {
   public void
   actionPerformed(ActionEvent evt) {
       btnBevestigActionPerformed (evt);
   }
}

MijnListener x = new MijnListener;
this.btnBevestig.addActionListener(x);
```

Anonieme klasse

```
this.btnBevestig.addActionListener(new
ActionListener() {
   public void
   actionPerformed(ActionEvent evt) {
       btnBevestigActionPerformed(evt);
   }
});
```

Custom events

`java.util.EventObject` is een basisklasse die speciaal gemaakt werd om andere event-informatieklasses op te baseren

Version Control Systems

= systemen om veranderingen aan een set van bestanden te beheren

- ook: revision control systems, source control systems

Belangrijkste taken

- ✓ bijhouden wat er gewijzigd werd
- ✓ bijhouden wie een wijziging heeftaangebracht
- ✓ bijhouden waarom een wijziging werd aangebracht

Nuttig in situaties waarin

- ✓ je wilt graag oudere versies van een document bijhouden
- ✓ je wilt de verschillen tussen meerdere versies van een bepaald bestand bijhouden
- ✓ je wilt de reden bijhouden waarom je bepaalde veranderingen hebt aangebracht
- ✓ je hebt een bestand verwijderd dat je terug wilt oproepen

Voordelen

Individueel gebruik

- alles wordt gebackupt
- alle wijzigingen en redenen voor wijzigingen worden bijgehouden
- je kan gemakkelijk teruggaan naar vorige versies
- je kan gemakkelijk een *sandbox* opzetten om in te experimenteren, zonder dat je bestanden in gevaar komen

Gebruik in team

- Gemakkelijke synchronisatie: elk teammember heeft toegang tot dezelfde content
- Het wordt gelogd wie welke wijzigingen heeft uitgevoerd.
- Conflicten detecteren: het systeem kan controleren of een gebruiker die nieuwe bestanden upload, niet ongewenst andere, bestaande bestanden overschrijft.

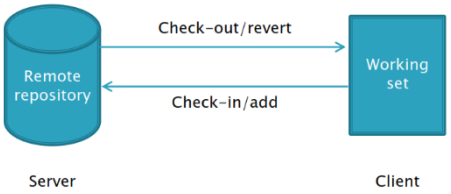
Terminologie

repository	een soort database waarin je mappen, bestanden en de geschiedenis ervan wordt bijgehouden (de core van het version control systeem)
working set/working copy	een lokale versie met mappen en bestanden op je harde schijf, deze bevat wijzigingen die mogelijks nog niet in de repository zitten
check-in (commit)	de wijzigingen van een working set in de repository invoegen
check-out (update)	de wijzigingen van de repository binnenhalen in de working set
tag/label	de huidige toestand beschrijven aan de hand van tekst zodat er later gemakkelijk naar kan verwezen worden
revert/rollback	overschrijven van bestanden in de working set met een specifieke versie die typisch afkomstig is van de repository
add	nieuwe bestanden van de working set invoegen in de repository

Soorten

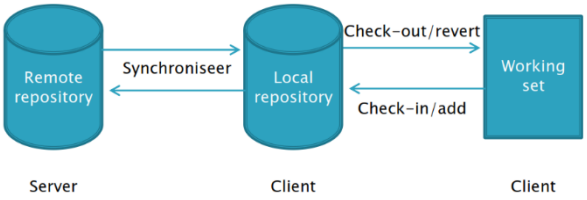
Gecentraliseerde systemen

- Gedeelde versies worden op centrale server bijgehouden
- Gebruiker maakt check-out van laatste versie project
- Na maken wijzigingen doet gebruiker check-in
- Voor check-out/check-in moet server beschikbaar zijn.
- Na een lange periode van offline werken krijg je één hele grote changeset.
- Backups maken van centrale systeem zijn wel gemakkelijker



Gedistribueerde systemen

- Er zijn 2 repositories
  - ✓ een lokale repository op de computer van de gebruiker
  - ✓ een remote repository op een centrale server



- De gebruiker kan altijd nieuwe versies inchecken en oudere versies bekijken op de lokale repository
- De gebruiker kan ten gepaste tijde zijn lokale repository synchroniseren met de remote repository

Hoe toegankelijk?

- Command line
- Shell integration (integratie met bestandverkenners als Windows Explorer en Mac Finder
- GUI tools
- IDE integration (o.a. Netbeans heeft standaard Git, SVN en Mercurial functionaliteiten ingebouwd)

Voorbeelden

Subversion, Git, TFS (Team Foundation Server), Mercurial

Het ontwikkelproces

1. Behoefteanalyse

Functieanalyse

= analyse van de omgeving waarin het systeem moet werken

- bv. beschrijving van processen, lijsten van gebruikers/profielen, maken van organogrammen, lijsten van de doelstellingen van het bedrijf, etc.

Definitiestudie

= analyse van wat het systeem allemaal moet doen

- beschrijving van de taken van het nieuwe systeem
- beschrijving van de grenzen van het systeem + communicatie
- door te vertrekken van bestaande systeem en te praten met stakeholders

2. Ontwerp

= wat het informaticasysteem allemaal moet omvatten

Functioneel ontwerp

= beschrijft taken waar het informaticasysteem bij te pas komt (sluit dicht aan bij behoefteanalyse)

- wie doet wat, wanneer en op welke manier?
- beschrijft verantwoordelijkheden van de mensen die met het systeem moeten omgaan

Technisch ontwerp

- beschrijving van de nodige apparatuur/hardware
  - → Bijbestellen? Aanpassen? Hergebruik?
- beschrijving van de nodige softwarepakketten
  - → bv. operating systems, gebruikersprogramma's, databases, webservers, pakketten op maat
- beschrijving van de realisatiefase
- beschrijving van de invoeringsfase

3. Realisatie

- Project realiseren (bv. installeren van hardware, schrijven, kopen, installeren van software)
- Testen en aanpassingen maken
- Documentatie maken en updaten bij veranderingen

4. Invoeren

- conversie van gegevens van oude systeem naar het nieuwe
  - → kan veel werk zijn, vooral als gegevens moeten worden bijgemaakt
  - → soms speciale software nodig, eventueel op maat bij te schrijven
- opleiden van gebruikers
- soms twee systemen (oud en nieuw) naast elkaar
  - → zo kort mogelijk houden → dubbel werk, synchronisatieproblemen, ...
- eindigt soms met een formele acceptatie
  - → de klant neemt na het testen en eventuele aanpassingen het nieuwe systeem in gebruik en verklaart dat aan alle eisen voldaan is

5. Onderhoud en beheer

= tijdens de normale uitbating van het nieuwe systeem

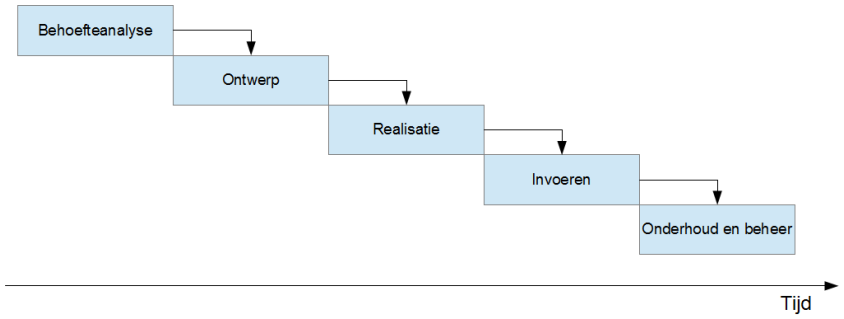
- kinderziekten oplossen (bv. fouten, nieuwe eisen/verwachtingen, etc.)
  - → eventueel terugkeren naar de vorige fase voor een gedeelte van het systeem
  - → hoe later nieuwe/andere behoeften ontdekt worden, hoe duurder om deze te implementeren
- normaal onderhoud (bv. change requests, reparaties, vervangingen, backups)
- documentatie up to date houden

Volgorde van het ontwikkelingsproces

Plan driven

= behoeften liggen op voorhand vast

ook: waterval / lineaire ontwikkelingsmodel



Change driven

- agile/iteratief model
- korte iteraties
- verkennende aanpak
- continu verbeteren
- wel high level analyse