Java Programming Language

Versie 1.0, maart '18

Introductie

- Voorstelrondje
- Agenda
- Dagindeling
- Cursusmateriaal
- Doel van deze cursus

Agenda dag 1 t/m 3

dag 1

- H1 Fundamenten van de taal Java
- H2 Datatypen, variabelen en methoden
- H3 Applicaties

dag 2

- H4 Overerving
- H5 Condities en herhalingsstatements
- H6 Packages en modifiers

dag 3

- H7 Constructors en overloading
- H8 Conversie, polymorfisme en interfaces

Agenda dag 4 en 5

dag 4

- H9 Arrays
- H10 Collecties
- H11 Garbage collection en foutafhandeling

dag 5

- H12 Lambda expressies
- H13 Date/Time

Dagindeling

- 8:45 -…… Begin cursusdag
  - Introductie
  - H1 Algemeen
  - H2 Hoofdstukinhoud
    - theorie
    - opdrachten maken
    - opdrachten bespreken
- 10:30-10:45 Koffiepauze
- 12:00-12:45 Lunchpauze
- 14:30-14:45 Koffiepauze
- …….-16:00 Einde cursusdag

Cursusmateriaal

- Cursusoverzicht

- mijn.vijfhart.nl
  - Samenvatting theorie
  - Bijlagen
  - Opdrachten

- Computer
  - java console
  - notepad++

Doel van deze cursus

- De syntax van Java leren kennen en gebruiken
- Object georiënteerd kunnen programmeren
- Voorbereiding op het OCA examen

H1 Fundamenten van de taal Java

Wat is Java?

- Een programmeertaal
  - Object georiënteerd
- Een verzameling packages met standaard klassen
- Een runtime omgeving (JVM)

procedureel programmeren

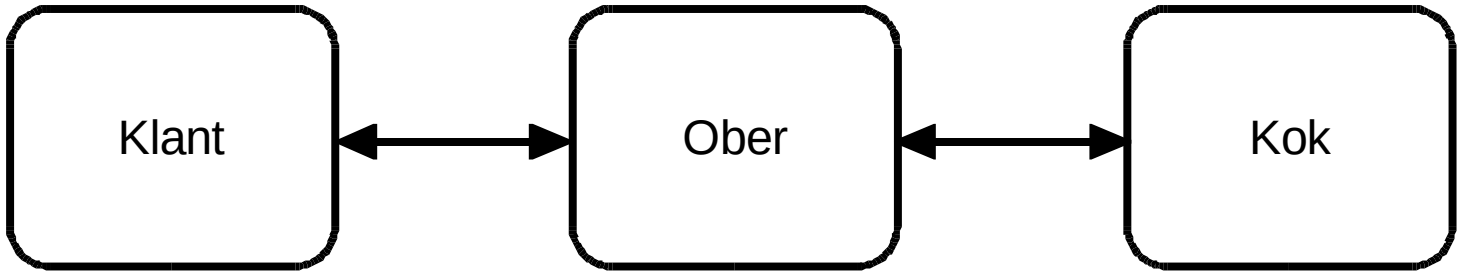
- welke taken moeten achtereenvolgens worden uitgevoerd?
- code voor taken die herhaald moeten worden komen in procedures of functies
- hoofdapplicatie roept deze functies en procedures aan
- voordeel: snel te bouwen
- nadeel: slecht te onderhouden

object georiënteerd programmeren

- welke data en taken zijn nodig?
- maak gespecialiseerde klassen om objecten mee aan te maken
- elk object is verantwoordelijk voor eigen gegevens en taken
- maak gebruik van bestaande functionaliteit door subklassen te maken
- breng in het hoofdprogramma de objecten in stelling
- laat de objecten hun taken uitvoeren
- objecten kunnen ook elkaar aan het werk zetten
- voordeel: goed onderhoudbaar, hergebruik van code
- nadeel: moeilijker te bouwen

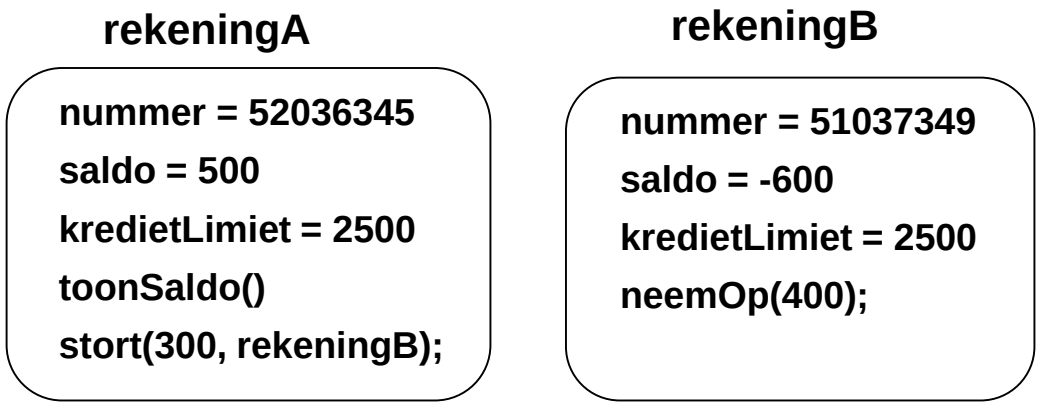
Separation of concerns

- Elke speler heeft z'n eigen verantwoordelijkheden
- Spelers weten van elkaar wat ze kunnen, en gebruiken deze kennis om met elkaar samen te werken

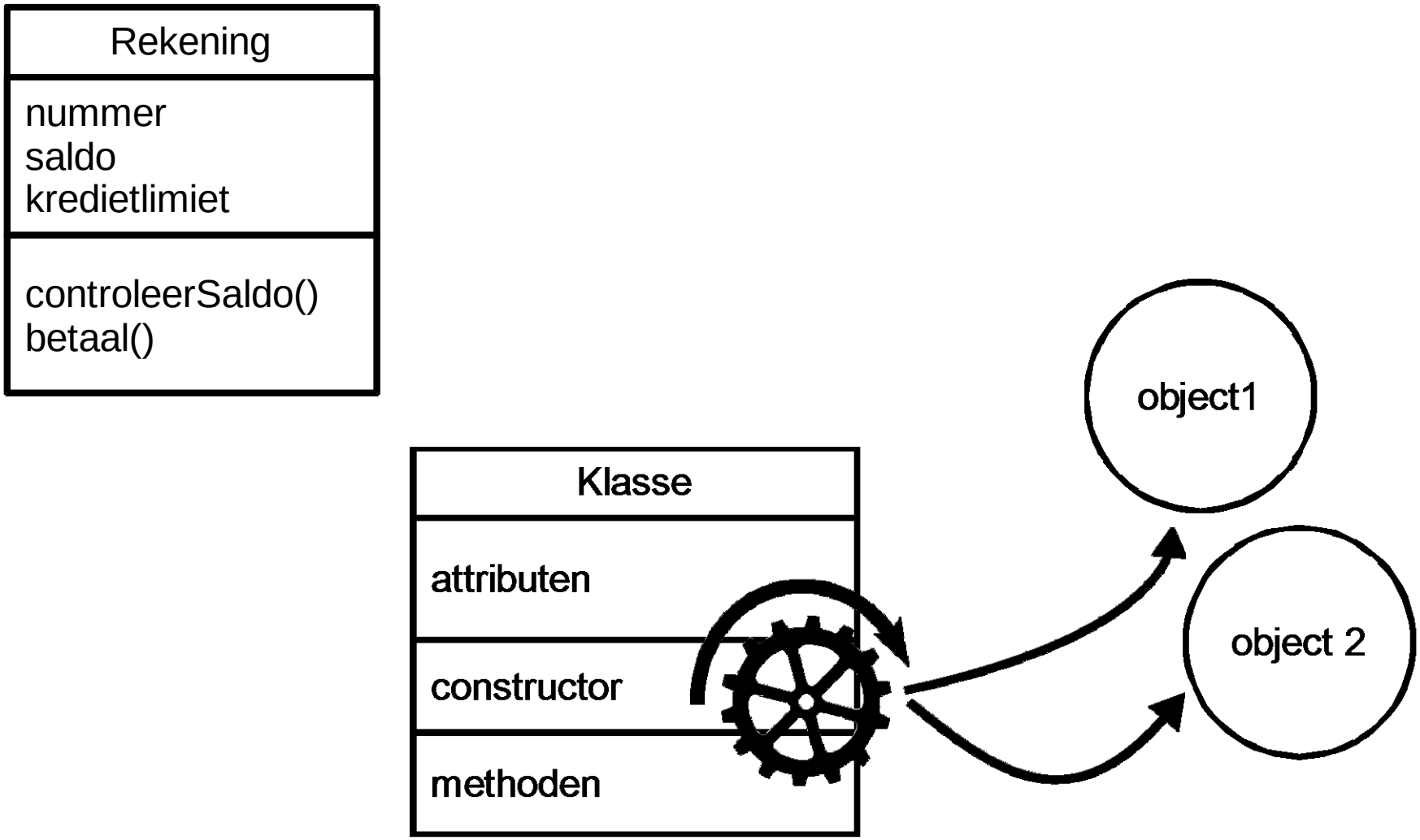


Data en gedrag

- Objecten bevatten gegevens (data)
- Objecten hebben gedrag:
  - methoden om eigen data te benaderen
  - methoden om andere objecten te benaderen



Objecten en klassen



Klasse - code voorbeeld

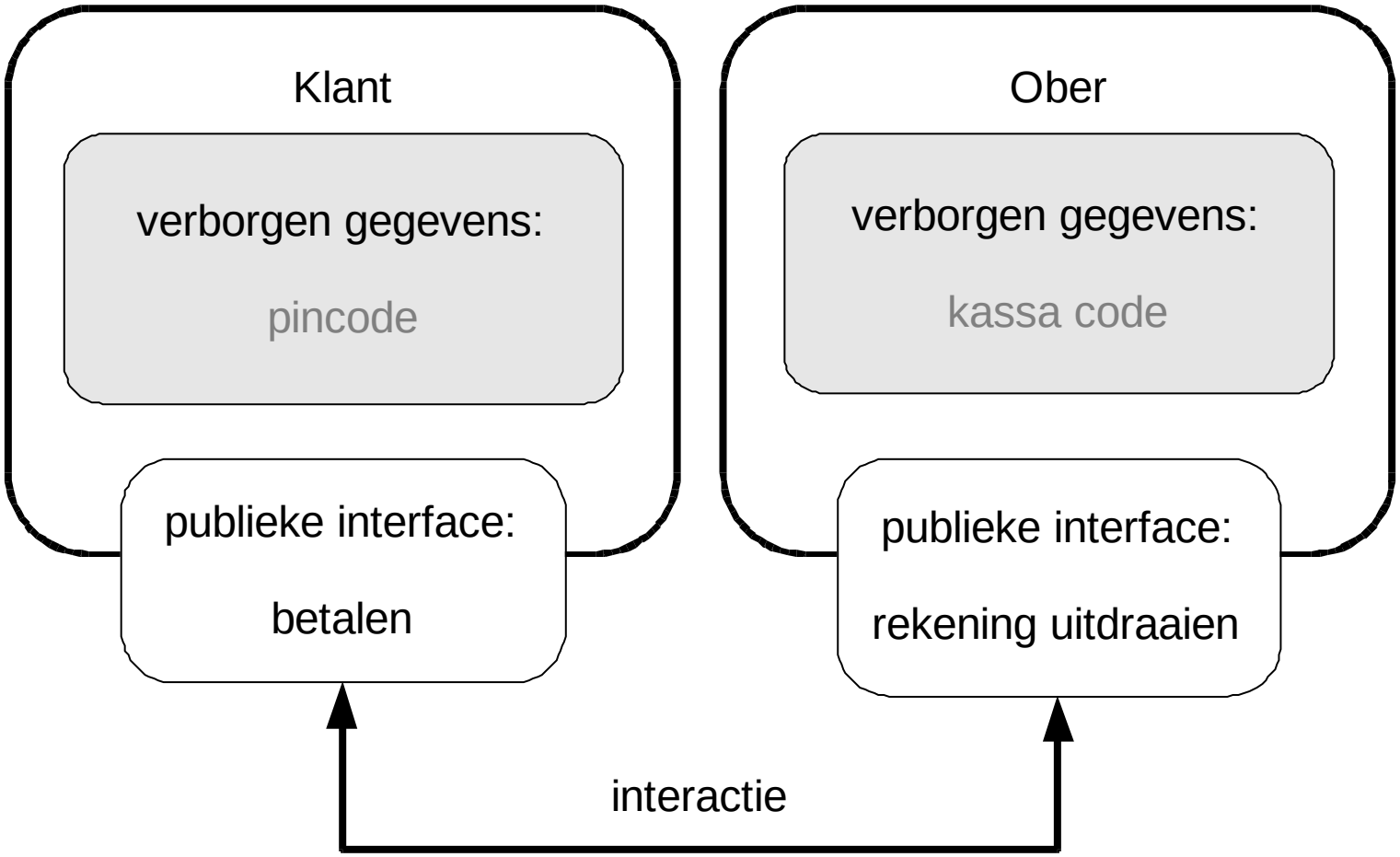
```
class Punt {
  int x;                              // attributen van de klasse
  int y;

  Punt(int a, int b) {                // een constructor
    x=a;
    y=b;
  }

  void verplaats(int a, int b) {  // een methode
    x = x + a;
    y = y + b;
  }
}
```

// volgorde bij voorkeur: attributen, constructors, methoden

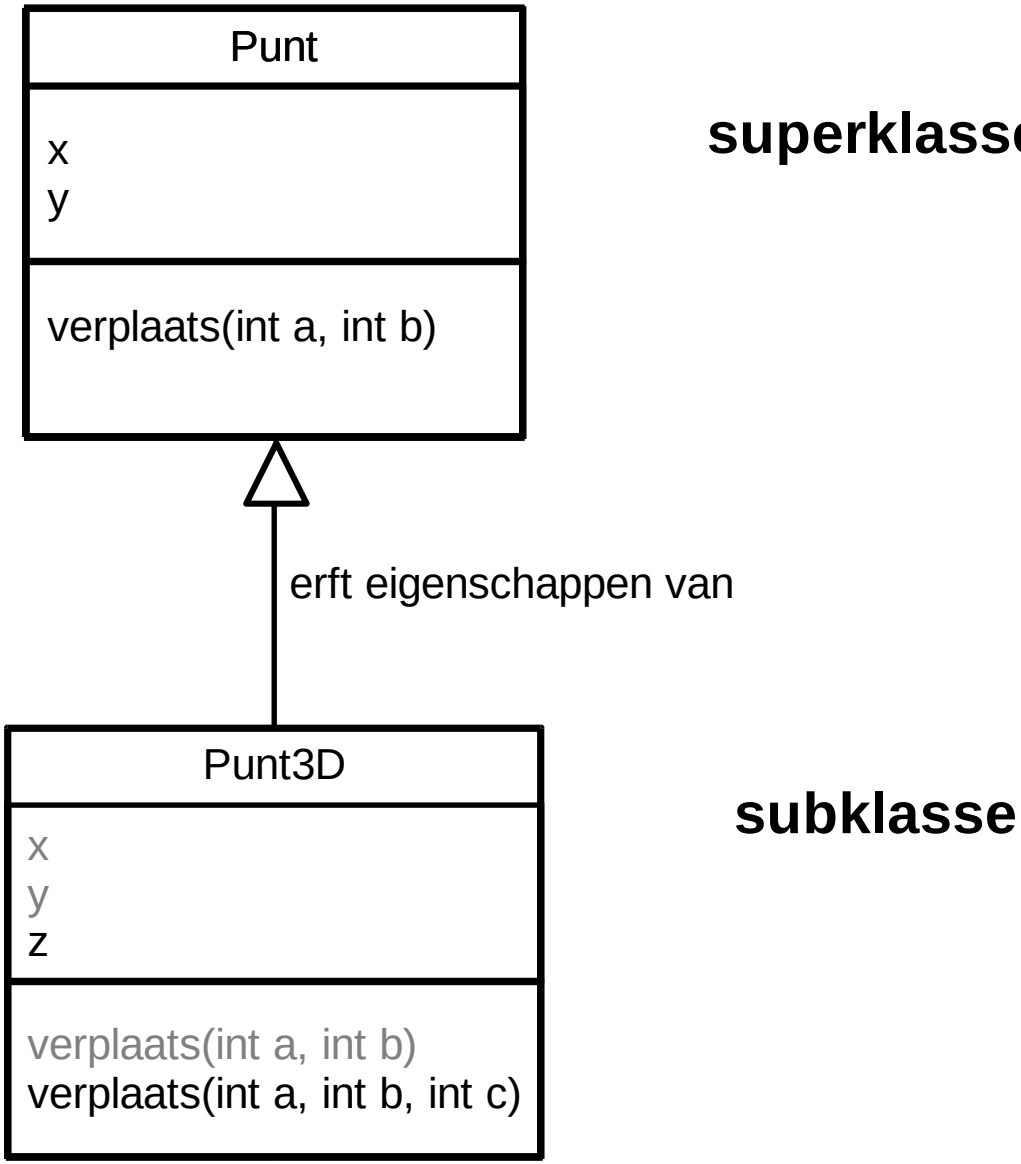
Objecten - code voorbeeld

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

    Punt p1 = new Punt(1,5);      // gebruik constructor om
    Punt p2 = new Punt (3,8);     // objecten p1 en p2 aan te maken

    p1.verplaats(4,-2);           // roep methoden aan
    p2.verplaats(3,1);            // om data te wijzigen

  }
}

// p1 heeft nu een x-waarde van 5 en een y waarde van 3
// p2 heeft nu een x-waarde van 6 en een y waarde van 9
```

Encapsulation



Overerving


Punt
x
y
verplaats(int a, int b)
superklasse
erft eigenschappen van
Punt3D
x
y
z
verplaats(int a, int b)
verplaats(int a, int b, int c)
subklasse

Java technologie

- Java Standard Edition (JSE)
- Java Enterprise Edition (JEE)
- Java Micro Edition (JME)

Java Development Kit (JDK) versies

Release-nummer	codenaam	Release-datum
JDK 1.1.4 - java 1	Sparkler	12 september 1997
JDK 1.1.5	Pumpkin	3 december 1997
JDK 1.1.6	Abigail	24 april 1998
JDK 1.1.7	Brutus	28 september 1998
JDK 1.1.8	Chelsea	8 april 1999
J2SE 1.2 - java 2	Playground	4 december 1998
J2SE 1.2.1	geen	30 maart 1999
J2SE 1.2.2	Cricket	8 juli 1999
J2SE 1.3	Kestrel	8 mei 2000
J2SE 1.3.1	Ladybird	17 mei 2001
J2SE 1.4.0	Merlin	13 februari 2002
J2SE 1.4.1	Hopper	16 september 2002
J2SE 1.4.2	Mantis	26 juni 2003
J2SE 5.0 (1.5.0)	Tiger	29 september 2004
Java SE 6 (1.6.0)	Mustang	11 december 2006
Java SE 7 (1.7.0)	Dolphin	28 juli 2011
Java SE 8 (1.8.0)	Geen codenamen meer	18 maart 2014

Tools van de JDK

- java (laadt java programma's en voert deze uit)
- javac (compiler)
- javadoc (genereert documentatie)
- jar (inpakken van packages)
- jdb (debugger)
- Java Class Library (standaard aanwezige klassen)
- apt (annotation processing tool)
- jconsole (grafische applicatie om performance te monitoren)
- jvisualvm (grafische schil om diverse JDK tools)
  - threads
  - profiler
  - monitor
  - etc..

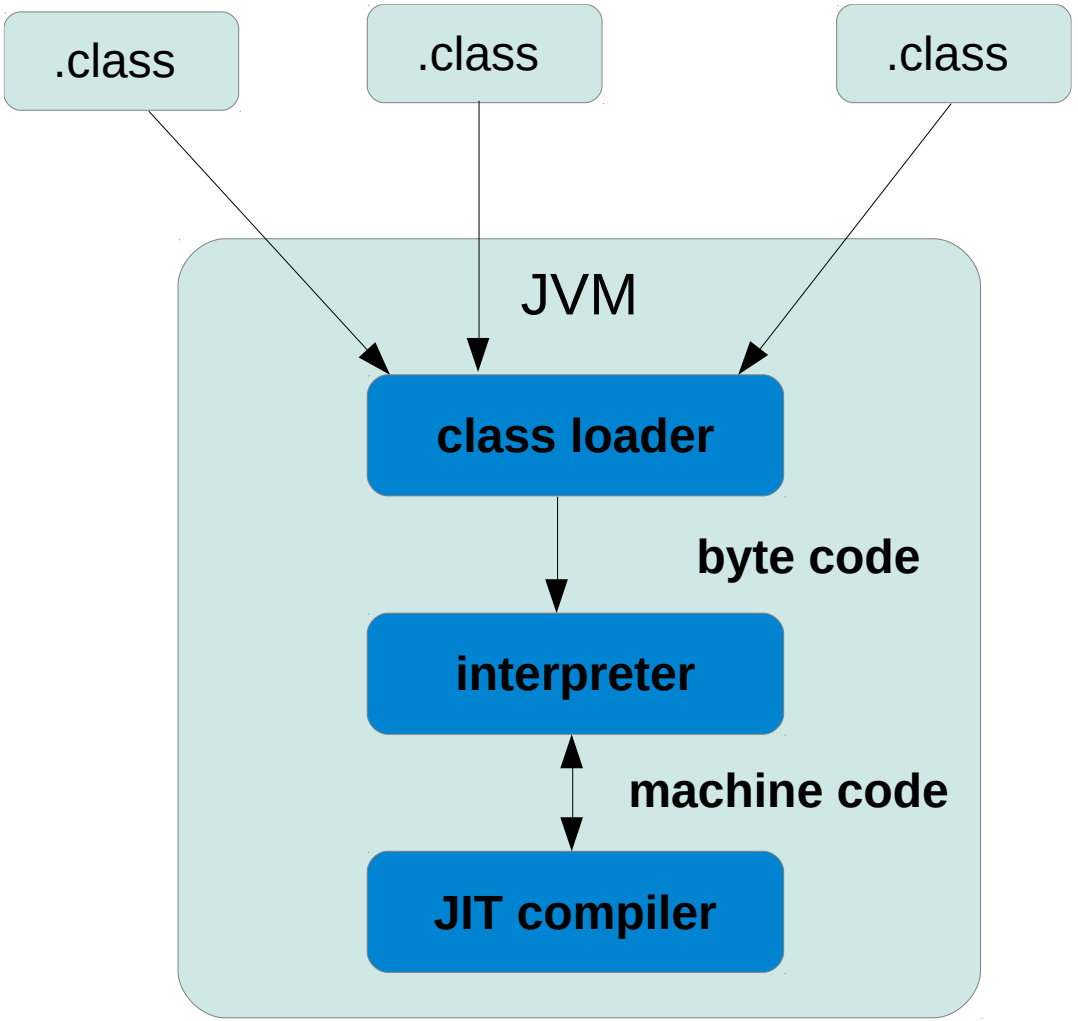
Van broncode naar byte code

- Java broncode staat in .java bestanden
- Met javac worden worden de .java bestanden gecompileerd
- Hierdoor ontstaan .class bestanden in byte code
- De .class bestanden kunnen worden gedraaid in een Java Virtual Machine (JVM).
- De JVM is specifiek voor het platform, .class bestanden kunnen in elke JVM worden gedraaid.

JIT compilatie

- Binnen de JVM laadt de class loader de .class bestanden
- De interpreter leest de byte code en voert deze uit
- De Just in Time (JIT) compiler genereert ondertussen machine code
- Als code fragmenten herhaald worden kunnen die in machine code worden uitgevoerd (sneller)

JIT compilatie - vervolg



Voorbeeld in broncode

Het volgende staat in Voorbeeld.java

```
class Voorbeeld
{
   public static void main(String args[])
   {
       System.out.println("Hello World!");
   }
}
```

compileren en uitvoeren

Compileren vanaf een DOS prompt:

```
javac Voorbeeld.java
```

Resultaat: Voorbeeld.class

Uitvoeren vanaf een DOS prompt:

```
java Voorbeeld
```

Uitvoer: Hello World!

Voorbeeld met twee klassen

Twee bestanden - Boodschap.java:

```
public class Boodschap {
   public void toonGroet() { /* toongroet is een methode */
      System.out.println("Hello World!");
   }
}
```

en HalloWereld.java:

```
public class HalloWereld { // HalloWereld is een uitvoerbare klasse
  public static void main (String args[]){
    Boodschap boodschap = new Boodschap();//  boodschap is een object
    boodschap.toonGroet();
  }
}
```

Structuur van een .java bestand

Klasse:

```
class KlasseNaam  {  /* klassenaam is hoofdlettergevoelig
                        en begint met hoofdletter */
    // code
}
```

code:

- attributen: bevatten de gegevens (***state***)
- constructors: maken objecten aan
- methoden: voeren code uit (***gedrag***)

Naamgeving

- Naam (identifier) van variabele, klasse, methode, etc. moet beginnen met een letter, een underscore of een $-teken
- Verder een onbeperkt aantal cijfers (0-9) of letters (A-Z, a-z)
- Een public class moet in een .java bestand met dezelfde naam worden gedeclareerd

Conventies:

- Gebruik duidelijke volledige namen, geen afkortingen
- Gebruik zelf geen $: bedoeld voor gegenereerde namen
- Begin een variabele-naam met kleine letter, verder CamelCase
- Begin een methode-naam met kleine letter, verder CamelCase
- Begin een klasse-naam met hoofdletter, verder CamelCase
- Constanten worden in hoofdletters geschreven, met underscores als scheidingstekens tussen woorden

Naamgeving: gereserveerde keywords

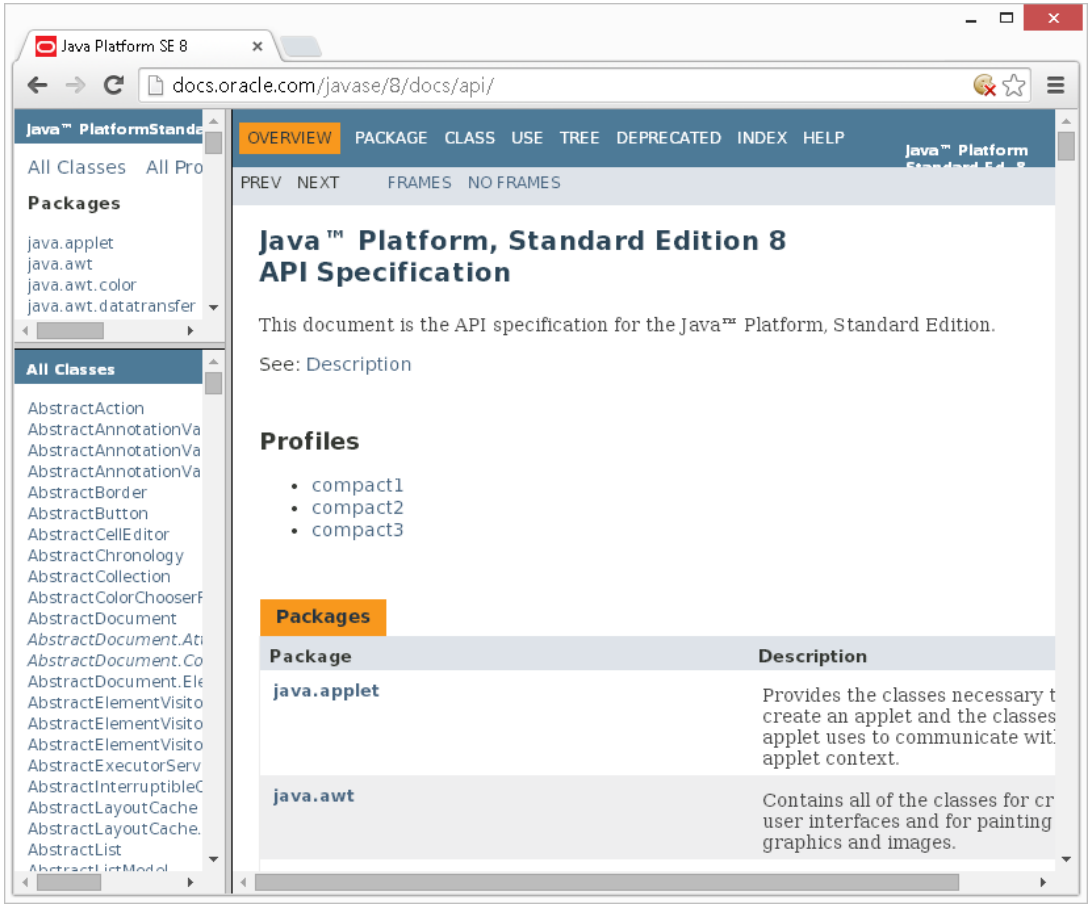
abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile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

*** worden niet gebruikt in java, maar zijn wel gereserveerd**

Ook null, true en false kunnen niet gebruikt worden als naam

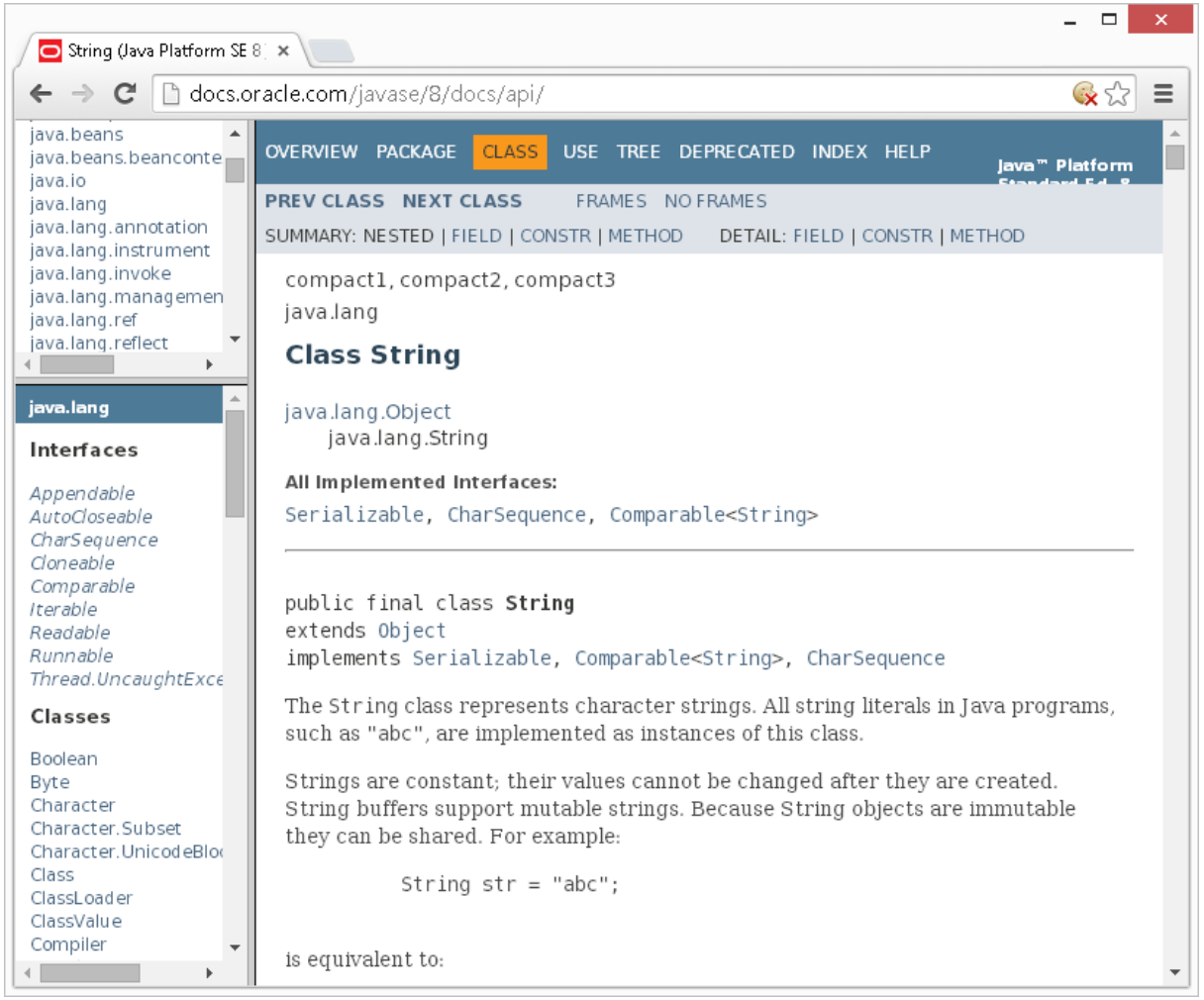
Java API specification

De documentatie bij JDK 8 staat op
http://docs.oracle.com/javase/8/docs/api/



Navigatie door de documentatie

- Klik links bovenaan op de package (bijvoorbeeld java.lang)
- Klik links onderaan op een klasse
- Rechts staat de documentatie van de klasse
  - Summary: Nested | Field | Constr | Method voor samenvatting
  - Detail: Field | Constr | Method voor uitgebreide documentatie



Opdrachten H1

H2: Datatypen, variabelen en methoden

Doel: gebruik kunnen maken van

- Java datatypen
- variabelen en constanten
- operatoren
- methoden

OCA boek: H1 en H2 t/m p66,
H3 p102 t/m 118 (String en StringBuilder)
H4 t/m p172 (begin van methods), p188 t/m 191

variabelen

- Variabelen hebben een datatype en een naam
- Worden voor gebruik gedeclareerd
- Kunnen direct worden geïnitialiseerd (van waarde voorzien)

Voorbeelden:

```
String naam;    // declaratie
int aantal = 0; // declaratie en initialisatie

// meerdere variabelen declareren en initialiseren
int getal1, getal2=10, getal3=20;
```

Datatypen

- Een object-variabele heeft het datatype van de betreffende klasse
- De variabele is een referentie die gekoppeld is aan een object
- De variabele kan leeg zijn: null
- voorbeelden:
  - Persoon a = new Persoon();
  - Auto b = new Auto();
  - String c = new String("voorbeeld");

- Andere variabelen hebben een primitief datatype
- Variabelen met een primitief datatypen alleen een waarde, geen attributen of methoden
- Primitieve variabelen kunnen niet leeg zijn
- voorbeelden:
  - int jaar = 2012;
  - boolean gehuwd = false;

Primitieve datatypen

Datatype	Grootte (bits)	Waarde	Bereik	Default waarde
byte	8	geheel getal	-128 t/m +127	0
short	16	geheel getal	-32.768 t/m +32.767	0
int	32	geheel getal	-2.147.483.648 t/m +2.147.483.647	0
long	64	geheel getal	$-(2^{63})$ t/m $+(2^{63}-1)$	0L
float	32	reëel getal	1.4E-45 t/m 3.4E38 (en negatief)	0.0F
double	64	reëel getal	4.9E-234 t/m 1.797E308 (en negatief)	0.0D
char	16	een Unicode karakter	0 t/m 65.535 (van '\u0000' t/m '\uffff')	\u0000 (null)
boolean	1	true of false	true of false	false

Integer datatypen

- Vier integer datatypen:
  - byte: 8 bits
  - short: 16 bits
  - int: 32 bits
  - long: 64 bits

- Een geheel getal is standaard een int
- Een long getal wordt aangeduid met een L achter het getal
- Voor byte en short bestaat zo'n aanduiding niet: kunnen gecast worden
- byte of short variabele kan direct met int waarde worden aangemaakt

Voorbeeld:

```
long milliseconden = 31536000000L;
// anders: error: integer number too large: 31536000000
byte aantal = 100; // byte variabele gevuld met een int
```

Voorbeeld: byte

- Elk getal wordt opgeslagen als een aantal bits
- Een geheel getal wordt “two’s complement” opgeslagen:
  - Negatieve getallen hebben hun eerste bit op 1 staan
  - Positieve getallen hebben de eerste bit op 0 staan
  - Het getal 0 heeft alle bits op 0 staan
  - Van x naar -x: alle bits omdraaien, 1 erbij optellen

0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	-1
1	1	1	1	1	1	1	0	=	-2
1	0	0	0	0	0	0	1	=	-127
1	0	0	0	0	0	0	0	=	-128

Decimaal, octaal, hexadecimaal

```
int decimaal = 100;
int octaal = 0124;
int hexadecimaal = 0xA19F;
int binair = 0b0110100;      // vanaf Java 7
```

- Integer getallen worden standaard decimaal weergegeven.
- Anders weergeven kan met een hulp-methode van een "wrapper" class: een klasse om primitieve waarden te bewerken.

voorbeeld:

```
System.out.println(Integer.toHexString(29));     // 1d
System.out.println(Integer.toOctalString(29));   // 35
System.out.println(Integer.toBinaryString(29)); // 11101
```

Floating point datatypen

- Gebroken (reële) getallen: float en double
- Standaard is een gebroken getal een double (dubbele precisie)
- float getallen worden aangeduid met F achter de waarde
- double getallen kunnen worden aangeduid met D achter de waarde
- Niet hoofdlettergevoelig: d of f mag ook
- 1.23E04 = 1.23 * 10 ^ 4 = 12300.0

float en double worden intern binair opgeslagen en geven bij berekeningen soms afrondingsfouten.

34.0/5.0 = 6.8
In machten van 2: 4 + 2 + 1/2 + 1/4 + 1/32 + ...
Binair: 1 1 0 . 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 (etc.)

double heeft dubbele precisie (8 bytes per getal)
ten opzichte van float (4 bytes per getal)

Underscores in getallen

- Lange getallen kunnen leesbaarder worden gemaakt met underscores als scheidingstekens (vanaf Java 7)
- Bij het compileren worden de scheidingstekens genegeerd
- Voorbeeld: `long grootgetal = 1_000_000_000L`
- Underscores mogen niet worden geplaatst:
  - aan het begin of eind van een getal
  - naast een punt (.) als decimaal scheidingsteken
  - direct voor het achtervoegsel L of andere type suffixen

Character

- Een karakter wordt in een char type opgeslagen.
- De waarde staat tussen enkele quotes,
- of als int getal
- 16 bits per karakter
- Notatie: char a = 'a', h='\u0068', e=0x0065, oct1=0154;

Bijzondere karakters worden met een escape sequence aangeduid:

'\n'	nieuwe regel
'\t'	tab
'\f'	form feed
'\r'	harde return
'\\'	back slash
'\''	enkele quote
'\"'	dubbele quote

Casting

- Numerieke typen kunnen naar elkaar gecast worden
- Dit gebeurt soms vanzelf: een klein getal past in een variabel van groter type:

```
float prijs = 4; double afstand = 13.2F; long aantal = 1234;
```

- Expliciet casten: zet het type tussen haakjes voor het getal

```
int geheelGetal = (int) 2.6;
```

- char en int kunnen (expliciet of impliciet) naar elkaar gecast worden:

```
char a = (char)97;
int  b = (int)'b';
char c = 99;
int d = 'd';
```

boolean datatype

- true of false
- variabele accepteert geen andere waarden

Literals

Een hard gecodeerde waarde wordt een literal genoemd

Voorbeelden:

```
int x = 10;
boolean test = true;
String naam = "Tijn";
final char zet = 'z';
```

Wrapper classes

- Wrapper classes geven extra functionaliteit bij primitieve typen
- o.a. om van String naar primitief datatype om te zetten

primitief datatype	wrapper class
boolean	Boolean
char	**Character**
byte	Byte
short	Short
int	**Integer**
long	Long
float	Float
double	Double

```
int x = Integer.parseInt("2"); // van String naar int
String s = String.valueOf(2);  // van int naar String
```

Scope en initialisatie

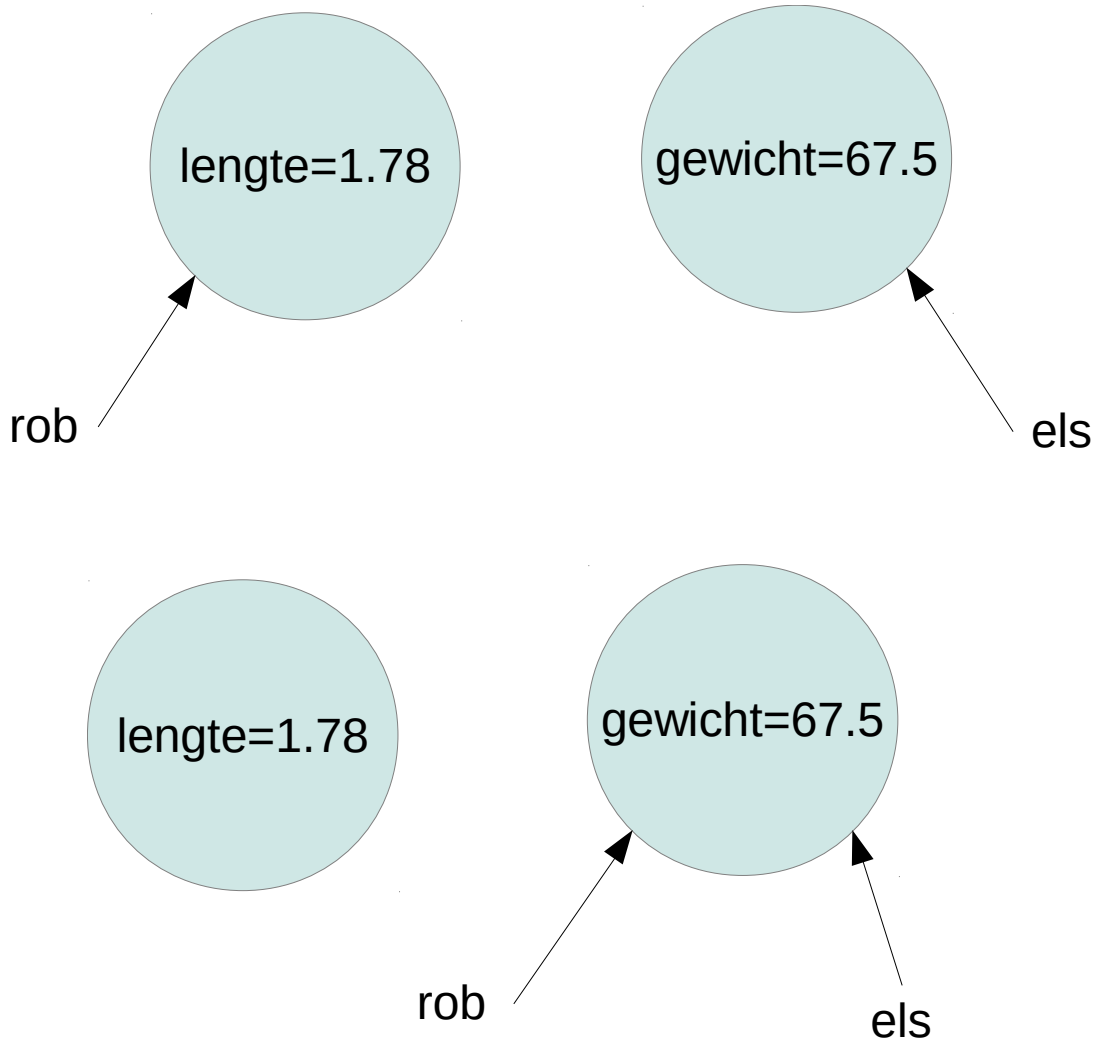
```
class Voorbeeld {
  static int globalVar; // Kan in meerdere methods en
                        // en van buitenaf gelezen worden.
                        // Niet geinitialiseerd?
                        // dan default waarde 0

  public static void main(String args[]){
    int localVar;       // moet worden geinitialiseerd
    localVar=1;         // voor gebruik
    System.out.println(globalVar < localVar);
  }
}
// uitvoer: true
```

Referentie datatypen

- niet-primitief type
- verwijst naar een object (instantie van een klasse)

```
class Persoon { double lengte, gewicht; } // attributen
...
{
  Persoon rob, els;          // declareren
  rob = new Persoon();       // initialiseren
  els = new Persoon();
  rob.lengte=1.78;           // waarden aan attributen toekennen
  els.gewicht=67.5;
  rob = els; // variabelen rob en els verwijzen
             // nu naar zelfde object
}
```

Referenties naar objecten



De String klasse

- String is een referentie type
- Waarde kan niet worden gewijzigd
- Wijziging van String variabele = aanmaken nieuw String object
- Kan worden geïnitialiseerd alsof het een primitief type is
- Wordt dan mogelijk uit de "String Constant Pool" gehaald

```
String tekst1 = new String("tekst"); // nieuw object

String tekst2 = "tekst"; // mogelijk nieuw object
String tekst3 = "tekst"; // anders uit de String Constant Pool

System.out.println(tekst1==tekst2); // false
System.out.println(tekst2==tekst3); // true
```

static variabelen vs instantie variabelen

Standaard globale variabele: instantie variabele
Met "modifier" static: klasse variabele.
Waarde van static variabele is voor alle objecten hetzelfde.

```
class Persoon {
  double lengte, gewicht;    // instance variabele
  static int aantalPersonen; // static variabele
  Persoon() {                // constructor
    aantalPersonen = aantalPersonen + 1;
  }
}
...
Persoon rob = new Persoon(); // aanroep constructor
Persoon els = new Persoon();
System.out.println(Persoon.aantalPersonen); // resultaat: 2
rob.aantalPersonen = 3;
System.out.println(els.aantalPersonen);     // resultaat: 3
```

Constanten

Variabelen die niet gewijzigd kunnen worden na initialisatie
Aangeduid met de modifier final
Vaak ook static: dan voor alle instanties gelijk

Voorbeeld:

```
final static double HALF_PI = Math.PI/2;

public final static int S = 0, M = 1, L = 2, XL = 3, XXL = 4;
public void checkMaat(int maat){
   if (maat == XL) {
     System.out.println("Extra Large");
   }
}
```

Nadelen: niet typesafe, geen namespace, niet informatief

Enums

Een enum type is een type met een vaste verzameling constanten.

```
public enum Maat { S, M, L, XL, XXL };
...
Maat mijnMaat = Maat.XL;
```

Voordelen:

- **type safe:** mijnMaat kan nu alleen S, M, L, XL, of XXL bevatten
- **namespace:** Maat.M is wat anders dan Letter.M
- **informatief:** geprinte waarden S, M, XL zijn duidelijker dan 0, 1, 3

Operatoren

Operatoren worden gebruikt voor:

- toekenning
- rekenkundige bewerkingen
- rekenkundige vergelijkingen
- bitwise operatoren
- logische (boolean) vergelijkingen
- samenvoegen van strings

Toekenning (assignment statement)

Toekennen: =

```
String taal = "Java";
int gros = 12 * 12;
double maandSalaris = jaarSalaris / 12;
```

Rekenkundige operatoren

+	optelling
-	aftrekking
*	vermenigvuldiging
/	deling
%	modulus (rest van een deling)

Strings samenvoegen

- Strings kunnen met + aan elkaar worden geknoopt.
- Impliciet worden daarbij numerieke waarden omgezet naar String

Voorbeeld:

```
System.out.println("6 + 4 = " + 6 + 4);
```

resultaat: 6 + 4 = 64

Oplossing:

```
System.out.println("6 + 4 = " + (6 + 4));
```

resultaat: 6 + 4 = 10

Toekenning voor/na bewerking

Operator	Beschrijving	Voorbeeld	Verklaring
++	postfix ophoging en toekenning	`y=x++;`	`y=x; x=x+1;`
	prefix ophoging en toekenning	`y=++x;`	`x=x+1; y=x;`
--	postfix aftrekking en toekenning	`y=x--;`	`y=x; x=x-1;`
	prefix aftrekking en toekenning	`y=--x;`	`x=x-1; y=x;`
+=	optelling en toekenning	`x+=y;`	`x=x+y;`
-=	aftrekking en toekenning	`x-=y;`	`x=x-y;`
*=	vermenigvuldiging en toekenning	`x*=y;`	`x=x*y;`
/=	deling en toekenning	`x/=y;`	`x=x/y;`
%=	modulus en toekenning	`x%=y;`	`x=x%y;`

Prefix en postfix bewerkingen

- Bij een prefix bewerking wordt eerst de bewerking uitgevoerd en daarna de expressie geëvalueerd.
- Bij een postfix bewerking wordt eerst de expressie geëvalueerd en daarna de bewerking uitgevoerd.

```
int x=1, y=1;
System.out.println(x++ +" "+ ++y + " " + x-- + " " + --y );
// uitvoer: 1 2 2 1
// waarde van x is hierna 1

x = x++;
System.out.println(x);

//uitvoer: 1
```

Samengestelde operatoren

- samengestelde operatoren: `+=` `-=` `*=` `/=` `%=`
- casten impliciet naar het type van te wijzigen variabele

```
int x = 3;
double y = 4.5;

x = x + y;
// compilatie fout: expected int, found double

x += y;
// gaat goed: x is nu 7
// zelfde als: x = (int) (x + y);
```

Vergelijkende operatoren

Operator	Beschrijving	Voorbeeld
==	is gelijk aan	`x == y`
!=	is ongelijk aan	`x != y`
<	is kleiner dan	`x < y`
>	is groter dan	`x > y`
<=	is kleiner dan of gelijk aan	`x <= y`
>=	is groter dan of gelijk aan	`x >= y`

== en != van primitieve typen:
vergelijkt of waarden gelijk zijn

== en != van referentie typen:
vergelijkt of beide variabelen refereren naar zelfde object

Vergelijking tussen referentie variabelen

Gebruik equals() in plaats van == bij vergelijking van referentie variabelen.

==	test of variabelen naar zelfde object verwijzen
equals()	test "gelijkwaardigheid" van de variabelen

```
String tekst1 = new String("tekst"); // nieuw object

String tekst2 = "tekst"; // mogelijk nieuw object
String tekst3 = "tekst"; // anders uit de String Constant Pool

System.out.println(tekst1==tekst2); // false
System.out.println(tekst2==tekst3); // true

System.out.println(tekst1.equals(tekst2)); // true
```

Logische operatoren

Bij vergelijking van meerdere boolean waarden

Operator	Beschrijving	Voorbeeld
&	AND	`z = x & y;`
&&	optimized AND	`z = x && y;`
|	OR	`z = x | y;`
||	optimized OR	`z = x || y;`
^	XOR	`z = x ^ y;`
!	NOT	`z = !x;`

optimized (of logical of conditional) AND:

false zodra een van de waarden false is,
overige waarden worden dan niet meer verwerkt

optimized (of logical of conditional) OR:

true zodra een van de waarden true is,
overige waarden worden niet meer verwerkt

Optimized AND en OR: voorbeeld

```
// gegeven object x
// wordt bewerkt in test1 en test2

if (test1(x) || test2(x)){
  doeIets();
}
```

Gevaar: als test1(x) true oplevert, wordt test2(x) niet meer uitgevoerd.

Wel handig:

```
if (x!=null && x.test() > 5 ){
  doeIets();
}
```

De ?: operator (conditional of ternary operator)

variabele = <boolean expressie> ? waarde : expressie

Als <boolean expressie> waar is, dan waarde, anders expressie

Voorbeeld: aantal wordt a als a groter is dan b, anders b.

```
aantal = a > b ? a : b;
```

Ook te combineren.
Voorbeeld: rank = -1 als a < b, +1 als a > b, anders 0

```
rank = a < b ? -1 : a > b ? 1 : 0;
```

Prioriteit van operatoren

volgorde	operator	beschrijving	richting
1	`[ ] ( ) .`	array index, methode aanroep, member toegang	links naar rechts
2	`++ --`	postfix ophoging, postfix verlaging	rechts naar links

3	`++ --`
	`+ -`
	`! ~`
		prefix ophoging en verlaging,
		positief, negatief
		logische NOT, bitwise NOT
			rechts naar links

4	`(type) new`	type cast, object aanmaken	rechts naar links
5	`* / %`	vermenigvuldiging, deling, modulus	links naar rechts
6	`+ - +`	optellen, aftrekken, concatenatie	links naar rechts
7	`<< >> >>>`	bitwise shift	links naar rechts

8	`< <=`
	`> >=`
	`instanceof`
		kleiner dan, kleiner dan of gelijk aan
		groter dan, groter dan of gelijk aan
		reference test
			links naar rechts

9	`== !=`	gelijkheid, ongelijkheid (waarde of reference)	links naar rechts
10	`&`	bitwise AND / Boolean AND	links naar rechts
11	`^`	bitwise XOR / Boolean XOR	links naar rechts
12	`|`	bitwise OR / Boolean OR	links naar rechts
13	`&&`	conditional (optimized) AND	links naar rechts
14	`||`	conditional (optimized) OR	links naar rechts
15	`? :`	conditional	rechts naar links

16	`= += -= *=`
	`/= %= &=`
	`^= |= <<=`
	`>>= >>>=`
		toekenning en samengestelde toekenningen	rechts naar links

Voorbeeld volgorde operatoren

```
int x=1, y=2;
System.out.println(x<2 || x>1 && y++<3 ? y : --x);
```

uitvoer: 2

uitleg: lees van links naar rechts, prioriteit uit tabel bepaalt haakjes (binnenste haakjes eerst) , let ook op optimized and en or

```
System.out.println((x<2) || ((x>1) && ((y++)<3)) ? y : (--x));
```

Richting bepaalt hoe gelijke operatoren worden geëvalueerd.

```
int x=1, y=2, z=3;
z += y += x += 4; // rechts naar links
System.out.printf("x=%s, y=%s, z=%s %n",x,y,z);
```

uitvoer: `x=5, y=7, z=10`

H2: opdracht 1 en 2

Methoden

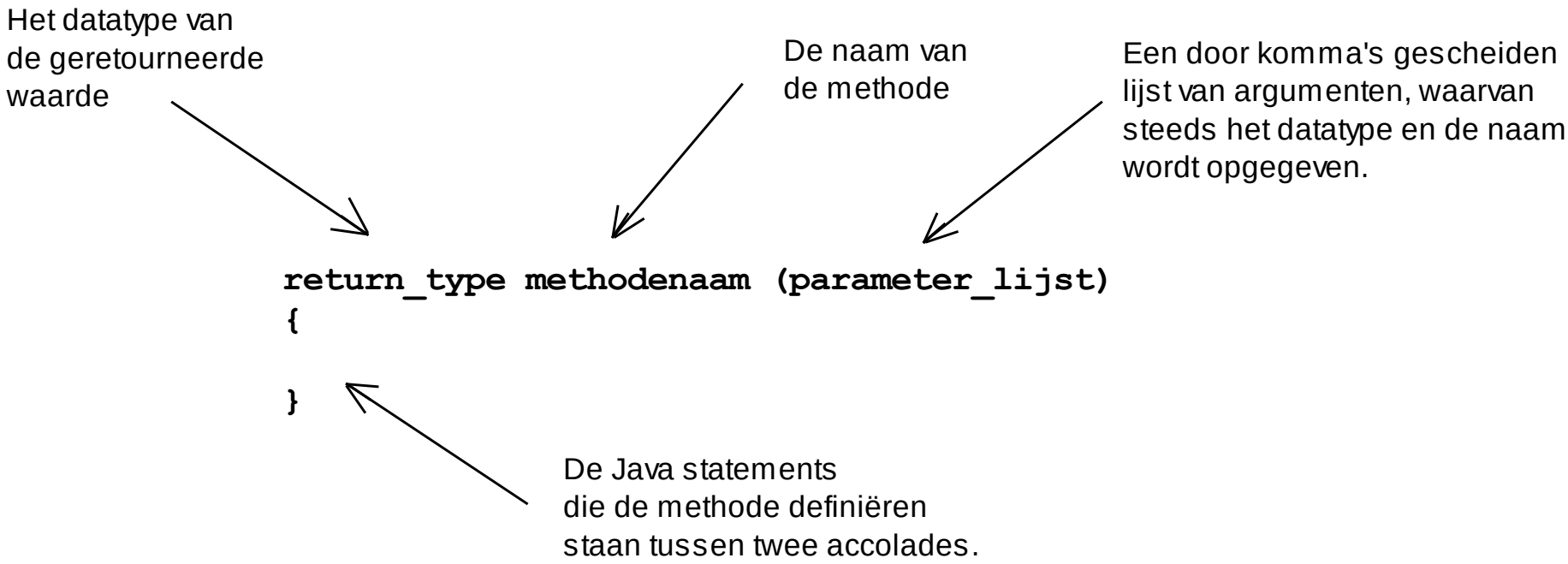
Syntax:

```
[<modifiers>] <return type> <methodenaam> ([<parameterlijst>])
{
   [<body>]
}
```

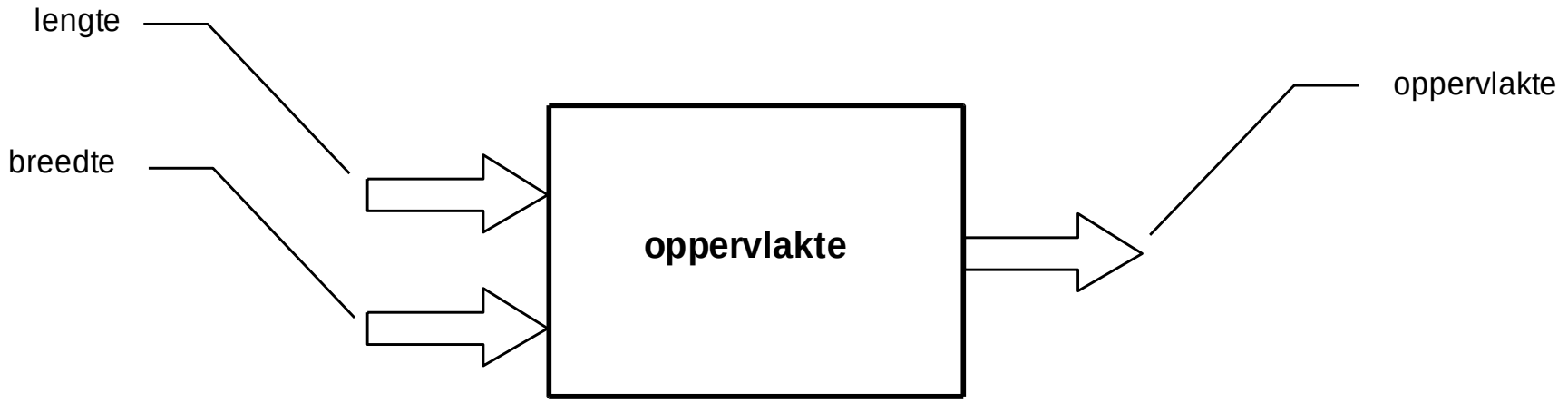
Voorbeeld:

```
public static void main (String args[]) {
   System.out.println("Hello World");
}
```

Structuur van een methode



Methode oppervlakte - proces

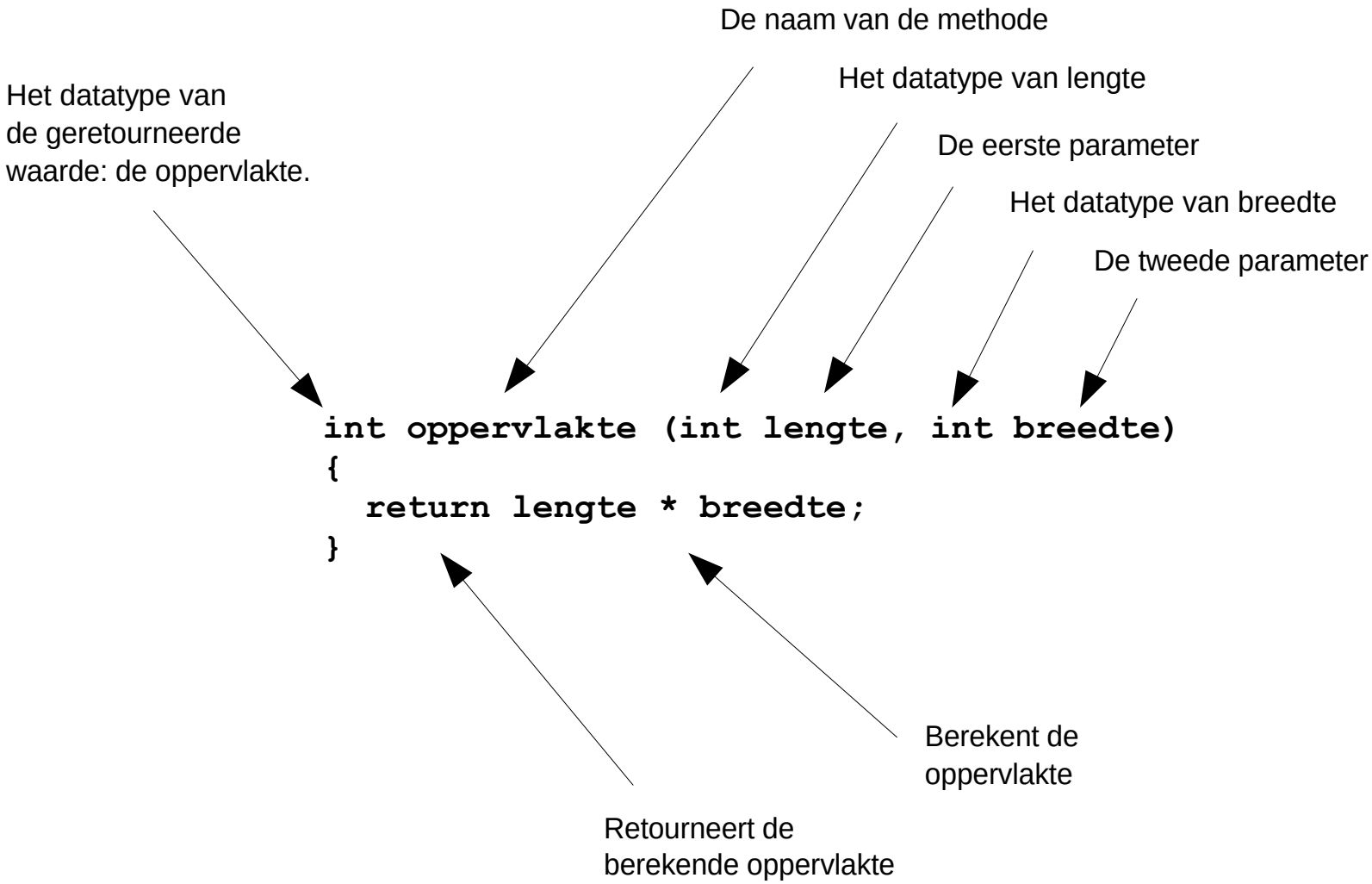


invoerwaarden
of parameters
of argumenten

berekening:
*oppervlakte = lengte * breedte*

geretourneerde waarde
of return waarde

Methode oppervlakte - in code



Return

- Met return <waarde> geven we een waarde van het opgegeven type terug
- Een return statement in een methode maakt ook dat de methode wordt verlaten
- Ook een void methode kan een return statement bevatten

Voorbeeld:

```
int oppervlakte (int lengte, int breedte) {
  return lengte * breedte;
}
```

Parameters

- Een methode kan nul of meer parameters bevatten.
- Waarden die aan methoden worden doorgegeven worden argumenten genoemd.
- Haakjes zijn verplicht, ook zonder parameters.
- Primitieve waarden worden als waarde doorgegeven: "pass by value"
  - De waarde van het argument wordt doorgegeven aan de parameter
  - Bij bewerking van de parameter blijft doorgegeven variabele ongewijzigd
- Objecten worden als referentie doorgegeven: "pass by reference"
  - De referentie van het argument wordt doorgegeven aan de parameter
  - Vanaf dat moment refereren de parameter en het argument naar hetzelfde object
  - Zolang de referentie in takt blijft (refereert naar zelfde object) worden wijzigingen binnen de methode toegepast op het onderliggende object

Pass by value - voorbeeld 1

```
   public static void main(String args[]) {
      int x = 5;
      test(x); // bewerking in methode
      System.out.println(x);
   }

   static void test(int waarde) {
     waarde++;
   }

// Uitvoer: 5
```

Pass by value - voorbeeld 2

```
   public static void main(String args[]) {
      int x = 5;
      x = test(x); // toekennen van returnwaarde van methode
      System.out.println(x);
   }

   static int test(int waarde) {
     waarde++;
     return waarde;
   }

// Uitvoer: 6
```

Pass by reference - voorbeeld 1

```
class Foo { int bar=1; }

public static void main (String args[]) {
  Foo foo = new Foo();
  bewerk(foo);
  System.out.println(foo.bar);
}

static void bewerk(Foo f) {
  // foo en f verwijzen naar zelfde object
  f.bar=0;
}

// Uitvoer: 0
```

Pass by reference - voorbeeld 2

```
class Foo { int bar=1; }

public static void main (String args[]) {
  Foo foo = new Foo();
  bewerk(foo);
  System.out.println(foo.bar);
}

static void bewerk(Foo f) {
  // foo en f verwijzen naar zelfde object
  f = new Foo(); // f verwijst nu naar nieuw object
  f.bar=0;
}

// Uitvoer: 1
```

Static methoden versus instantie methoden

- zonder static is het een instantie methode
- instantie methoden zijn bedoeld om:
  - data (state) van eigen object te bewerken, of
  - data (state) van eigen object beschikbaar te stellen

- met static is het een klasse methode
- static methoden hebben geen instantie nodig, kunnen direct vanuit de klasse worden aangeroepen
  - hulpmethoden, alleen gebruikt om het resultaat van de methode, niet om de state van een object te bewerken of te tonen
  - voorbeeld: Math.round(); Auto.getMerken();

Methoden - gebruik

- instantie-methoden: aanroepen met <instantie>.<methodenaam>
- instantie-methoden binnen eigen klasse: aanroepen met this.<methodenaam> of met <methodenaam>
- static methoden: aanroepen met <klassenaam>.<methodenaam>
- static methoden mogen ook met <objectnaam>.<methodenaam> worden aangeroepen (af te raden)

static en instantie methoden - voorbeeld

```
class Cijfer {
 private int waarde;  private static int hoogste;
 void setWaarde(int x){
    waarde = x;
    hoogste = x>hoogste? x: hoogste;
 }
 int getWaarde(){ return waarde; }
 static int getHoogste(){ return hoogste; }
}

class Test {
  public static void main (String args[]){
    Cijfer a = new Cijfer(); a.setWaarde(3);
    Cijfer b = new Cijfer(); b.setWaarde(5);
    System.out.println(a.getWaarde());        // 3
    System.out.println(Cijfer.getHoogste()); // 5
    System.out.println(a.getHoogste());       // 5
  }
}
```

Methoden van String

`char charAt(int x)`	karakter op positie x
`boolean endsWith(String s)`	true als String eindigt op String s
`boolean equals(Object o)`	true bij dezelfde tekstinhoud
`boolean equalsIgnoreCase (String s)`	true bij dezelfde tekstinhoud (niet hoofdlettergevoelig)
`int length()`	lengte van de string
`String replace (CharSequence c1, CharSequence c2)`	vervangt in String alle voorkomens van de tekenreeks c1 door tekenreeks c2.
`boolean startsWith(String s, int startpositie)`	true als String vanaf startpositie begint met opgegeven String s
`boolean startsWith(String s)`	true als String begint met opgegeven String s
`String substring(int x [, int y])`	String vanaf positie x (0 based) tot y (zonder y: tot eind van de String
`String toLowerCase()`	String in kleine letters
`String toUpperCase()`	String in hoofdletters
`String trim()`	verwijdert whitespace vooraan en achteraan

Methoden van StringBuilder (en StringBuffer)

Hiermee kunnen we tekstobjecten maken die gewijzigd kunnen worden.
StringBuilder is over het algemeen sneller dan StringBuffer.
StringBuffer is thread-safe, StringBuilder niet.
Let op: boolean equals(Object o) is in StringBuilder niet overschreven!

`StringBuilder append(String s)`	voegt String s aan het eind toe
`StringBuilder delete(int start, int eind)`	verwijdert de tekenreeks van start tot eind
`StringBuilder deleteCharAt(int p)`	verwijdert karakter op positie p
`StringBuilder insert(int p, String s)`	voegt String s toe op positie p
`StringBuilder reverse()`	zet tekenreeks in omgekeerde volgorde
`void setLength(int x)`	kapt af of vult met spaties aan tot lengte x
`void setCharAt(int p, char c)`	wijzigt karakter op positie p

append() en insert() kunnen ook andere gegevens toevoegen (overloaded)
method call chaining: `sb.append("a").insert(1,'b').deleteCharAt(0);`

Methoden van Math (static)

`static double abs(double x)`	absolute waarde van x
`static double floor(double x)`	grootste gehele getal kleiner of gelijk aan x
`static double ceil(double x)`	kleinste gehele getal groter of gelijk aan x
`static double pow(double x, double y)`	x tot de macht y
`static long round(double x)`	afgeronde waarde als long
`static double random()`	geeft random getal >= 0.0 en < 1.0
`static double sqrt(double x)`	wortel van x
`static double min(double x, double y)`	kleinste van de twee
`static double max(double x, double y)`	grootste van de twee

Meeste methoden komen ook voor met ander datatype voor de invoer en uitvoer.

Voorbeeld:

```
float uitkomst = (float)Math.round(3.14159F * 100) / 100;
```

Communiceren tussen objecten (1)

```
class Artikel {
  String naam; double prijs;
  Artikel(String n, double p){
     naam=n;
     prijs=p;
  }
  double totaal(int aantal){
    return prijs * aantal;
  }
}
```

Communiceren tussen objecten (2)

```
class Kassa {
   StringBuilder bon = new StringBuilder();
   double totaalPrijs;

   public void scan(Artikel a, int aantal){
     bon.append(a.naam).append(" ")
        .append(aantal).append(" x ")
        .append(a.prijs).append(": ")
        .append(a.totaal(aantal)).append("\n");
     totaalPrijs = totaalPrijs + a.totaal(aantal);
   }
   public void afrekenen(){
     bon.append("Totaal: ").append(totaalPrijs);
     System.out.println(bon.toString());
     bon.setLength(0);
     totaal=0;
   }
}
```

Communiceren tussen objecten (3)

```
class ArtikelApp {                          // het hoofdprogramma
   public static void main(String args[]){
      Artikel a = new Artikel("pen", 1.5);
      Artikel b = new Artikel("potlood", 0.8);
      Kassa kassa = new Kassa();
      kassa.scan(a, 2); // gebruikt totaal() van a
      kassa.scan(b, 4); // gebruikt totaal() van b
      kassa.afrekenen();
   }
}
// uitvoer:
pen 2 x 1.5: 3.0
potlood 4 x 0.8: 3.2
Totaal: 6.2
```

H2: opgave 3 en 4

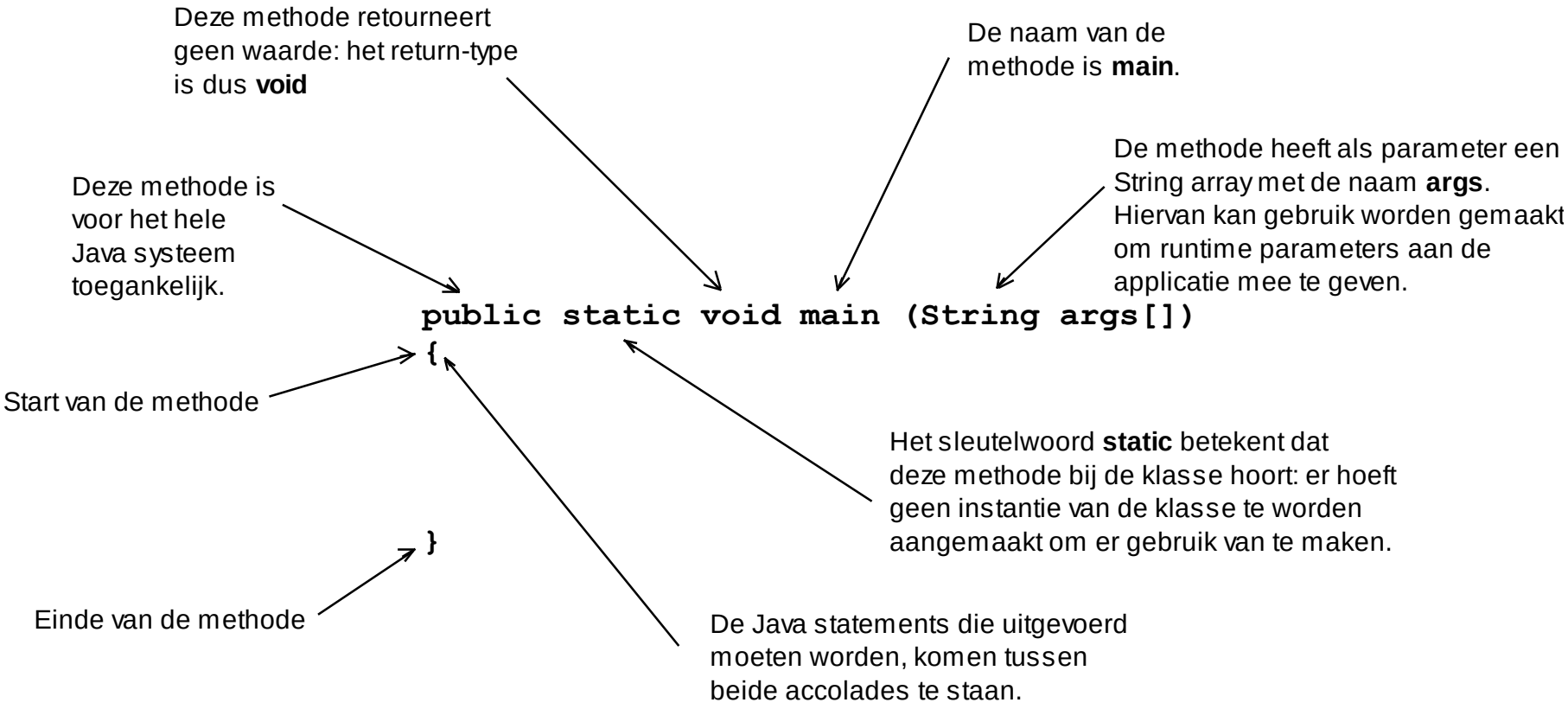
H3: applicaties

Doelen:

- onderdelen van een applicatie kennen en kunnen gebruiken
- argumenten kunnen meegeven en opvangen

OCA boek p6,7,8

De main methode



Signatuur en return type van methode main

- public
- static
- void
- main(<parameter>)

<parameter>:

- String <naam>[] of
- String[] <naam>

meestal wordt "args" als parameternaam gebruikt, hoeft niet

main methode - static context

```
class MijnApp {
  int x=3;
  public static void main(String args[]){
    System.out.println("De waarde van x is " + x);
  }
}
```

```
MijnApp.java:5: error: non-static variable x cannot be referenced
from a static context
    System.out.println("De waarde van x is " + x);
```

main methode - static context 2

```
class MijnApp2 {
  static int x=3;
  public static void main(String args[]){
    System.out.println("De waarde van x is " + x);
  }
}

class MijnApp3 {
  int x=3;
  public static void main(String args[]){
    MijnApp3 ma = new MijnApp3();          // object aanmaken
    System.out.println("De waarde van x is " + ma.x);
  }
}
// De waarde van x is 3
```

Argumenten doorgeven

```
class Feliciteer {
  public static void main (String jarigen[]){
    StringBuilder melding = new StringBuilder("Gefeliciteerd ");
    melding.append(jarigen[0]);
    melding.append("!");
    System.out.println(melding);
  }
}
```

Alternatief:

```
 public static void main (String jarigen[]){
    System.out.printf("Gefeliciteerd %s!", jarigen[0]);
 }
```

Uitvoeren: `java Feliciteer "jarige Job"`

Resultaat: `Gefeliciteerd jarige Job!`

Meerdere argumenten

```
class Test {
     public static void main (String deelnemers[]){
       System.out.printf("Deelnemers: %s, %s en %s",
        deelnemers[0], deelnemers[1], deelnemers[2]);
     }
}
```

Uitvoeren: `java Test Kwik Kwek Kwak Donald`

Resultaat: `Deelnemers: Kwik, Kwek en Kwak`

Uitvoeren: java Test Ernie Bert

Resultaat:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at test.main(Test.java:4)
```

Numerieke gegevens

- Ook numerieke gegevens worden als String doorgegeven
- Voor elk primitief type bestaat een "wrapper class", met hulp-methoden, o.a. om String om te zetten naar primitief type

```
class Vermenigvuldig {
 public static void main (String getal[]) {
    double x = Double.parseDouble(getal[0]);
    double y = Double.parseDouble(getal[1]);
    System.out.printf("%s * %s = %s", x, y, (x * y));
  }
}
```

Uitvoeren: `java Vermenigvuldig 3.4 2.1`

Resultaat: `3.4 * 2.1 = 7.14`

H3 opgaven

H4 Overerving

Doelen:

- concept van overerving begrijpen en toepassen
- toString() van Object overschrijven
- beperkingen van single parent systeem kennen
- gebruik kunnen maken van abstracte klassen

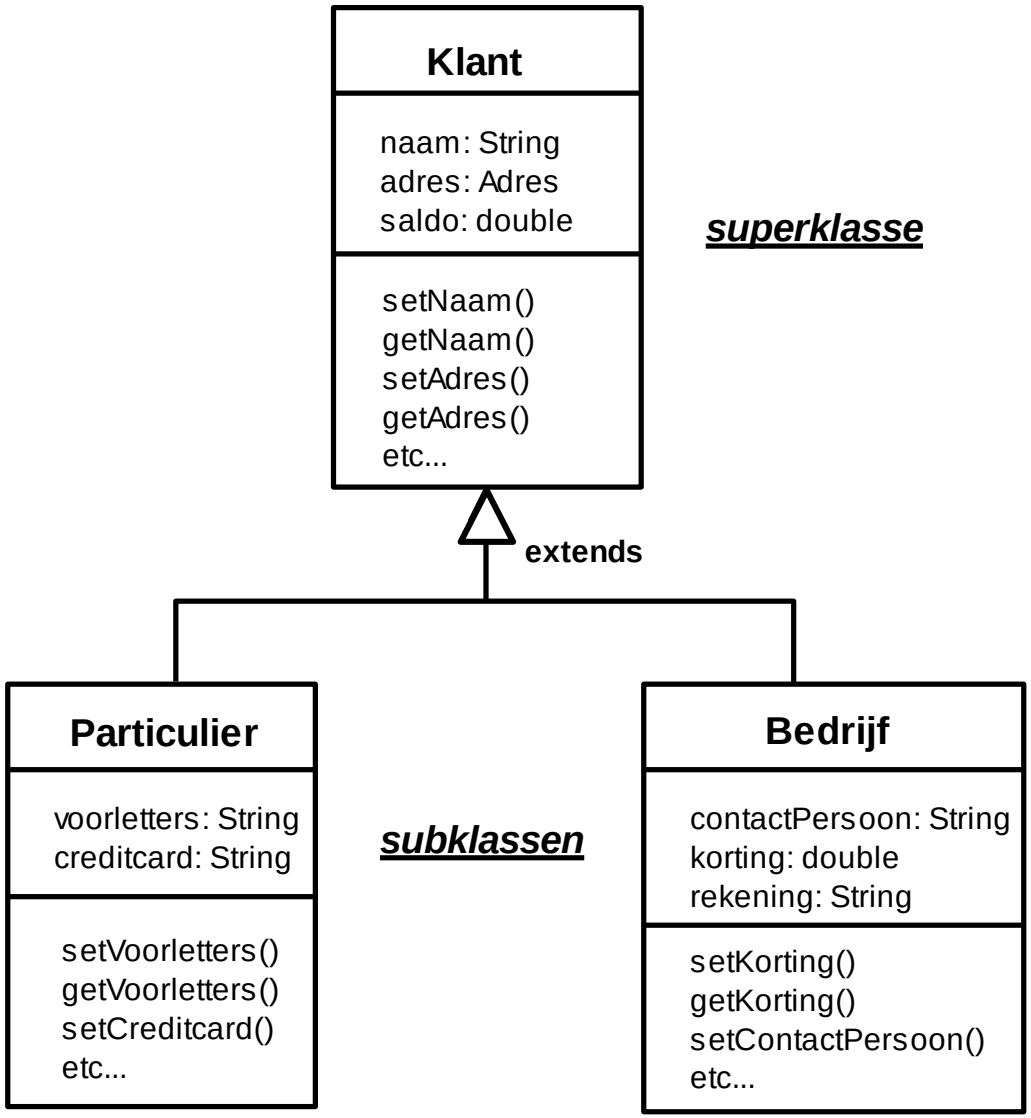
OCA boek: begin van H5 t/m p238, p246 t/m p251, p259 t/m 265

Subklassen

- Subklasse erft methoden en attributen van de superklasse
- <subklasse> *is een* <superklasse>
- in .java bestand:

```
class MijnSubklasse extends MijnSuperklasse {
        ...
}
```

Klassediagram



Single parent systeem

- Een klasse kan maar één superklasse hebben
- Levert soms problemen op
  - Een StudentAssistent *is een* Student
  - Een StudentAssistent *is* soms ook *een* Docent
- Oplossing: een klasse kan meerdere interfaces implementeren

```
class StudentAssistent extends Student implements Docent {
   ...
}
```

Methoden overschrijven

- Subklasse erft methoden van superklasse
- Code kan in subklasse worden overschreven, om een subklasse-specifieke implementatie van de methode mogelijk te maken
- Annotatie @Override kan worden gebruikt: compiler geeft een foutmelding als te overschrijven methode niet bestaat in superklasse

```
class Lamp {
  void zetAan() {
    drukOpAanUitKnop();
  }
}

class FietsLamp extends Lamp {
  @Override
  void zetAan() {
    activeerDynamoEnGaFietsen();
  }
}
```

Overriding en overloading

- Een overschreven methode zorgt ervoor dat de methode van de superklasse niet meer wordt gevonden (**overriding**)
- Om een methode te overschrijven moet de signatuur gelijk zijn
- Bovendien moet het return type hetzelfde zijn, of een subtype van het return type van de methode in de superklasse (**covariant return types**)
- Signatuur:
  - de naam van de methode
  - het aantal en de volgorde van de parameters
  - het datatype van de parameters
  - NIET: het return type

- Binnen een klasse(-hiërarchie) kunnen meerdere methoden met dezelfde naam voorkomen (**overloading**)
- De signatuur moet dan verschillen: aantal parameters en/of datatype van de parameters

Overriding en overloading voorbeeld

```
class Artikel {
  String naam; double prijs;
  double totaal(int aantal){
    return prijs * aantal;
  }
}
class ArtikelMetBTW extends Artikel {
  int BTW;
  @Override
  double totaal(int aantal) { // andere implementatie in subclass
    return prijs * aantal * (1 + BTW * 0.01);
  }
  // overload: extra optie
  double totaal(int aantal, boolean inclusief){
    return inclusief ? totaal(aantal): super.totaal(aantal);
  }
}
```

De klasse Object - methode toString()

- java.lang.Object is de uiteindelijke superklasse van alle klassen
- bevat methoden die bedoeld zijn om te overschrijven
- toString() geeft een label van het object, standaard als volgt opgebouwd:

  <klassenaam>@<hexadecimale hashcode>

- Als object wordt weergegeven met System.out.println(object) wordt de toString() methode van het object gebruikt
- Overschrijven: gebruik (delen van) state van het object om een duidelijke representatie van het object te genereren

```
public String toString(){
  return String.format("Persoon met lengte %s en gewicht %s.",
                       lengte, gewicht);
}
```

Abstracte klassen

- Van abstracte klassen kunnen geen instantie worden aangemaakt
- Bedoeld om subklassen van te maken
- Abstracte klassen kunnen abstracte methoden bevatten
- Deze abstracte methoden moeten in een concrete subklasse worden overschreven
- Abstracte methoden hoeven niet te worden overschreven in een abstracte subklasse

Abstracte klassen: voorbeeld

```
abstract class Voertuig {
  abstract void starten();
  abstract void stoppen();
}

class Fiets extends Voertuig {
  void starten(){
     zetPedalenInBeweging();
  }
  void zetPedalenInBeweging(){ // code }
}
```

foutmelding bij compileren: **`Fiets is not abstract and does not override abstract method stoppen() in Voertuig`**

`void stoppen(){}` toevoegen is genoeg om te laten compileren (lege implementatie)

H4 opgaven

H5 Condities en herhalingsstatements

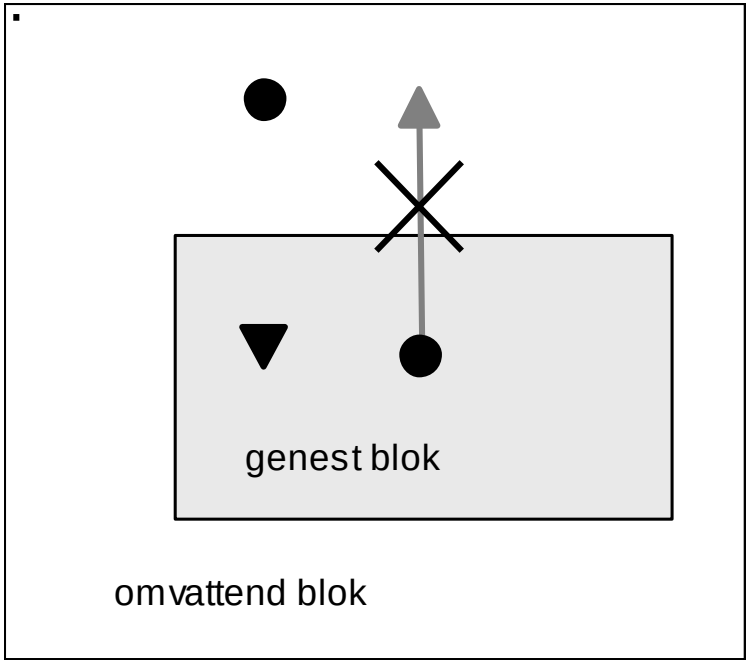
Doelen:

- **if** en **else** kunnen toepassen
- het **switch** statement kunnen gebruiken
- gebruik kunnen maken van **for**, **do** en **while** loops
- waar nodig **break** en **continue** kunnen inzetten

OCA boek: p66 t/m eind H2

Blokken

- Statements worden gegroepeerd in blokken: Java code tussen accolades
- Blokken kunnen worden genest
- Binnen elk blok kunnen lokale variabelen worden aangemaakt
- Lokale variabelen zijn alleen binnen dat blok (en geneste blokken) beschikbaar



if en else: conditional statements

syntax:

if (<boolean expressie>) <statement 1> [else <statement 2>]

of

if (< boolean expressie >) {<blok 1>} [else {<blok 2>}]

Elk blok kan ook if en else bevatten

Voorbeeld if en else

```
if (var1 == var2)
{
  System.out.println("var1 en var2 zijn gelijk");
}
else
{
  System.out.println("var1 en var2 zijn ongelijk");
}
```

Let op: een puntkomma ; geldt als leeg statement

```
if (var1 == var2);
  System.out.println("var1 en var2 zijn gelijk");
```

Deze melding wordt nu ook gegeven als var1 ongelijk is aan var2

if - pas op voor assignment ipv conditie

```
boolean test = false;
if (test = true)
  System.out.println("De test is gelukt");
else
  System.out.println("De test is niet gelukt");

// uitvoer: De test is gelukt

int x = 0;
if (x = 1)
  System.out.println("x = 1");

// compilatie error: incompatible types
//                   required: boolean
//                   found: int
```

Geneste if statements - voorbeeld 1

```
class Kies
{
  public static void main(String args[])  {
    int test = Integer.parseInt(args[0]);
    if(test>10) {
      System.out.println("groter dan 10");
    }
    else  {
      if(test>5) {
        System.out.println("tussen 6 en 10");
      }
      else {
        System.out.println("5 of kleiner");
      }
    }
  }
}
```

Geneste if statements - voorbeeld 2

```
class Kies
{
  public static void main(String args[])  {
    int test = Integer.parseInt(args[0]);
    if(test>10)
      System.out.println("groter dan 10");
    else if(test>5)
      System.out.println("tussen 6 en 10");
    else
      System.out.println("5 of kleiner");
  }
}
```

Het switch statement (conditional)

- beslisboom afhankelijk van de waarde van een int, char (impliciet gecast naar int), byte, short, Enum (vanaf Java 5), of String (vanaf Java 7)
- binnen switch worden mogelijke waarden opgegeven
- als waarde is gevonden wordt alle code daaronder uitgevoerd tot een eventuele break - dus ook code die bij andere waarden hoort!
- is de waarde niet gevonden, dan wordt de code achter default uitgevoerd (default is optioneel)

```
switch (expressie) {
  case waarde1:
     statement1;
  case waarde2:
     statement2;
     ...
  default:
    statement3;
}
```

Switch statement - voorbeeld

```
int dag = 3;
switch(dag) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
    System.out.println("Werkdag");
    break;
  case 6:
  case 7:
    System.out.println("Weekend");
    break;
  default:
    System.out.println("Geen bestaande dag van de week");
}
```

Switch - voorbeeld 2

```
public enum Score { UITSTEKEND, ZEER_GOED, GOED, RUIM_VOLDOENDE,
VOLDOENDE, TWIJFELACHTIG, ONVOLDOENDE, RUIM_ONVOLDOENDE, SLECHT,
ZEER_SLECHT}

public static void main(String[] args) {
        Score examenScore      = Score.GOED;
        switch (examenScore) {
                case UITSTEKEND:
                System.out.println("mooi zo");
                        break;
                case ZEER_GOED:
                case GOED:
                System.out.println("dat kan beter");
                        break;
                default:
                System.out.println("ga je schamen");
        }
}
```

Switch met String (vanaf Java 7)

- Een switch statement kan nu op basis van een String waarde worden uitgevoerd
- Een switch statement op basis van Strings kan meestal efficiënter worden gecompileerd dan een if-then-else beslisboom op basis van Strings
- Elke case bevat een String waarde, of een final variabele die (direct) in de String constant pool is geplaatst, dus "waarde" ipv new String("waarde")

```
public static void main(String args[]) {
  final String maandag = "maandag";
  switch(args[0].toLowerCase()) {
    case maandag: case "dinsdag": case "woensdag":
    case "donderdag": case "vrijdag":
    System.out.println("Werkdag");  break;
    case "zaterdag": case "zondag": System.out.println("Weekend");
    break;
    default: System.out.println("Geen bestaande dag van de week");
  }
}
```

Loops

- while loop — iteration statement
- do while loop — iteration statement
- for loop — iteration statement
- break, return en continue — transfer of control statement

While loop

while (<boolean expressie>) <blok>

- wordt doorlopen zolang <boolean expressie> waar is
- binnen blok moet <boolean expressie> op een bepaald moment false worden, anders wordt blok oneindig doorlopen
- oneindige loop: **`while (true) {...}`**

```
int teller = 1;
while (teller<=10) {
  System.out.println("teller = " + teller);
  teller++;
}
```

Vergelijk:

```
while (teller<=10)
  System.out.println("teller = " + teller);
  teller++;
```

Do while loop

do <blok> while (<boolean expressie>);

- Blok wordt nu in ieder geval één keer doorlopen, ook als <boolean expressie> initieel false is.
- Blok wordt herhaald zolang <boolean expressie> true is

Wat komt hier uit?

```
public static void main(String args[]){
  int x=0;
  do{
    System.out.println(x++);
  }while(x==1);
  System.out.println(x);
}
```

for loop

for (<initialisatie expressie>;
<boolean expressie>;
<update / iteratie expressie per loop>) <blok>

- <initialisatie expressie> wordt alleen aan begin van de loop uitgevoerd
  - meerdere variabelen van zelfde type mogen hier worden geïnitialiseerd, gescheiden door komma's
  - scope van de variabelen die hier worden aangemaakt is het blok van de for loop
- <expressie per loop> wordt na elke loop uitgevoerd
  - mag uit meerdere statements bestaan, gescheiden door komma's
- als <boolean expressie> nog waar is, wordt het blok herhaald
- minder gevaar op oneindige loop
- oneindige loop: `for(;;){...}`

for loop - voorbeeld

```
for(int i=1, j=5; i<j; i++, j--)
{
  System.out.println("i = " + i);
  System.out.println("j = " + j + "\n");
}
```

Resultaat:

```
i = 1
j = 5

i = 2
j = 4
```

for loop - genest voorbeeld

```
for (int i = 1;i<=100;i+=10) {
      for(int j = i; j<i+10; j++) {
        if (j<100){
          System.out.print(" ");
          if (j<10)
            System.out.print(" ");
        }
        System.out.print(" " + j);
      }
      System.out.println();
}
```

Geneste for loop - resultaat

```
 1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30
31  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60
61  62  63  64  65  66  67  68  69  70
71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90
91  92  93  94  95  96  97  98  99 100
```

false in if en loops

- Voor de compiler is een conditie die altijd false is vaak een foutsituatie
- compiler error bij:
  - while(false){...}
  - for(...;false;...){...}

- geen compiler error bij:
  - if(false){...} // veel gebruikt bij debuggen
  - do{...}while(false); // blok wordt nu 1 keer doorlopen

break

- break zorgt ervoor dat een loop of een switch wordt verlaten
- als code binnen loop volgt op break, moet break conditioneel worden gemaakt
- anders error: unreachable statement

```
public static void main(String[] args) {
  for(;;){
    System.out.println("test a");
    if(true){break;}
    System.out.println("test b");
  }
  System.out.println("test c");
}
```

Uitvoer:

```
test a
test c
```

break met label

- optioneel kan achter break naar een label worden verwezen
- hiermee kan uit een geneste loop worden gesprongen naar een hoger niveau dan het vorige

```
mijnLabel:
for(;;)
{
  while(true)
  {
    if(conditie)
    break mijnLabel;
  }
}
// hier gaan we verder na break mijnLabel:
```

return

- return zorgt ervoor dat de methode wordt verlaten
- als return in een loop staat wordt dan dus ook de loop verlaten

```
public static void main (String args[])
{
  for (int i=0;i<args.length;i++)    // voor alle parameters
  {
    System.out.println(args[i]);     // toon parameter
    if (args[i].equals("stop"))      // als parameterwaarde is "stop"
     return;                         // stop de main() methode
  }
}
```

continue

- vanaf continue wordt de rest van de code binnen een loop overgeslagen
- er wordt direct doorgegaan met de volgende stap in de loop

```
for (int jaar = 2000; jaar <3000; jaar ++)
{
  if (jaar%100 == 0 && jaar%400 != 0)
    continue;
  if (jaar%4 == 0)
    System.out.println(jaar + " is een schrikkeljaar");
}
```

continue met label

- achter continue kan naar een label worden verwezen
- de gelabelde loop wordt dan direct vervolgd

Voorbeeld: toon alle opgegeven woorden, behalve woorden
met twee dezelfde karakters achter elkaar

```
woorden:
for(int i=0;i<args.length;i++){ // van alle opgegeven woorden ...
   for(int j=0;j<args[i].length()-1;j++){ // alle letters doorlopen
      char deze = args[i].charAt(j);
      char volgende = args[i].charAt(j+1);
      if(deze==volgende) continue woorden;
   }
   System.out.println(args[i]); // wordt soms overgeslagen
}
```

Opgaven H5

H6 Packages en modifiers

Doelen

- loose coupling en high cohesion begrijpen en toepassen
- klassen kunnen onderbrengen in packages
- klassen uit andere packages kunnen gebruiken
- toegangcontrole kunnen regelen met modifiers
- eigenaardigheden van static members kennen
- weten wat modifier final doet voor members en klassen

OCA boek: H1 p9 t/m 15 (packages)
H4 p173 t/m 185 (access modifiers)
H4 p187 en 188 (static imports)

design overwegingen

- gestreefd moet worden naar lage afhankelijkheid tussen klassen: **loose coupling**
- gestreefd moet worden naar transpantie, door implementatie-details in de code af te schermen: **encapsulation**
- encapsulation is een belangrijke manier om coupling te beperken
- gestreefd moet worden naar een hoge samenhang tussen data en de methoden die die data gebruiken of bewerken. Een klasse moet doen waar het voor bedoeld is, niet meer en niet minder: **high cohesion**
- high cohesion zorgt ook voor loose coupling: een klasse die verantwoordelijk is voor eigen gedrag en gegevens, heeft daarmee weinig afhankelijkheden met andere klassen

packages

- Een package is een geheel van gerelateerde klassen
- Definieert de "name space" van een klasse: hierbinnen moeten namen uniek zijn
- Puntnotatie, van algemeen tot specifiek
- Standaard packages: java.lang, java.util, java.io, java.sql, etc.
- Naam conventies:
  - gebruik kleine letters
  - gebruik naam van eigen domein in omgekeerde volgorde
  - voorbeeld: com.vijfhart.cursus
  - gebruik underscores voor samengestelde namen
  - voorbeeld: com.vijfhart.java_cursus_ocaj
  - package namen die beginnen met java en javax zijn gereserveerd
- "Fully quallified name" van een klasse is <package>.<klasse>
  - voorbeeld: java.util.Date

Standaard aanwezige packages

package	omschrijving
java.util	collecties, datums, tijd, internationalisatie, event-model, diversen
java.util.stream	Klassen die functionele bewerkingen via lambda expressies op stromen van elementen mogelijk maken
java.util.function	Functionele interfaces die het gebruik van lambda expressies mogelijk maken (vanaf Java 8)
java.util.concurrent	Hulpklassen voor het programmeren van applicaties die gebruik kunnen maken van meerdere processoren
java.io en java.nio	Input en output functionaliteit voor files, streams en serialisatie
java.time	Nieuwe package voor datums, tijden, momenten en periodes (vanaf Java 8)
java.text	bewerken van tekst, datums, getallen en boodschappen onafhankelijk van de gebruikte taal.
java.sql	JDBC: Java Database Connectivity. Voor het benaderen van een database.
javafx	Klassen en intefaces voor het bouwen van GUI applicaties

- standaard packages beginnen met java
- java extensies beginnen met javax

Eigen klassen onderbrengen in packages

- Bovenaan in het .java bestand staat *package* <packagenaam>; bijvoorbeeld:

```
package com.vijfhart.cursus;
```

- package statement geldt voor alle klassen van het .java bestand
- package structuur komt overeen met directory structuur van het .java bronbestand
- zet .java bestanden met package com.vijfhart.cursus in een directory com\vijfhart\cursus
- compileren vanuit de parent directory van com\vijfhart\cursus:

```
javac com\vijfhart\cursus\*.java
```

- alternatief:
  - zet alle .java bestanden van een package in een aparte directory
  - geef daarin het commando: javac -d . *.java
  - -d . : .class bestanden inclusief package structuur worden in huidige directory (.) gegenereerd

- Alle .class bestanden komen nu in com\vijfhart\cursus te staan

bronbestanden en klassebestanden scheiden

- geef met -sourcepath aan waar .java bronbestanden staan
- geef met -d aan waar .class bestanden gegenereerd moeten worden

Voorbeeld aanmaken com.vijfhart.cursus.Voorbeeld.class:

- maak directory src aan, met daarin com\vijfhart\cursus
- ga in parent directory van src staan

```
javac -sourcepath src src\com\vijfhart\cursus\Voorbeeld.java
      -d classes
```

- nu ontstaat het bestand classes\com\vijfhart\cursus\Voorbeeld.class

```
java -classpath classes com.vijfhart.cursus.Voorbeeld
```

import

- Standaard zijn klassen binnen zelfde directory en klassen van java.lang direct beschikbaar
- Klassen uit andere packages kunnen direct worden gebruikt door fully qualified naam te gebruiken, zoals:

```
java.util.Date date = new java.util.Date();
```

- Om ze zonder packagenaam te benaderen moeten de klassen uit een package worden geïmporteerd:

```
import java.util.Date;
import java.sql.*;
```

- Klassen uit sub-packages moeten apart worden geïmporteerd:

```
import java.awt.*;
import java.awt.event.*;
```

- Klassen uit de default package kunnen niet worden geïmporteerd
- import statement geldt voor alle klassen in een .java bestand

import - voorbeeld

```
package com.vijfhart.cursus;
import java.util.*;
import java.sql.*;

class DateTest {
  Date datum;
}
```

Foutmelding bij compileren:

```
reference to Date is ambiguous, both class java.sql.Date in java.sql
and class java.util.Date in java.util match
```

Oplossing: extra import regel toevoegen:

```
import java.util.Date;
```

jar: bestanden inpakken

- packages worden in .jar bestanden ingepakt
- .jar bestanden kunnen in CLASSPATH worden gezet
- ingepakte .class bestanden zijn dan direct te benaderen (hoeven niet te worden uitgepakt)
- inpakken, uitpakken, updaten met de tool jar

Inpakken van com.vijfhart.cursus.Voorbeeld in voorbeeld.jar:

```
jar cf voorbeeld.jar -C classes com
```

  - opties: c=create, f=file
  - hele directoryboom onder com wordt ingepakt
  - com staat in de subdirectory classes

jar - inhoud bestand bekijken

```
jar tf voorbeeld.jar

META-INF/
META-INF/MANIFEST.MF
com/
com/vijfhart/
com/vijfhart/cursus/
com/vijfhart/cursus/PackageApp.class
com/vijfhart/cursus/Test.class
```

MANIFEST.MF is het Manifest bestand - dit bevat algemene informatie over het jar bestand

jar bestand gebruiken

Uitvoeren van com.vijfhart.cursus.PackageApp in voorbeeld.jar:

```
java -classpath voorbeeld.jar com.vijfhart.cursus.PackageApp
```

Om het jar bestand zelf uitvoerbaar te maken, geven we in het manifest bestand aan welke klasse we als de applicatie-klasse beschouwen:

```
jar ufe voorbeeld.jar com.vijfhart.cursus.PackageApp
```

e = entry point

In MANIFEST.MF komt dan het volgende te staan:

```
Manifest-Version: 1.0
Created-By: 1.7.0 (Oracle Corporation)
Main-Class: com.vijfhart.cursus.PackageApp
```

Hierna uitvoeren: `java -jar voorbeeld.jar`

Modifiers

Modifiers speciferen extra karakteristieken van een klasse, methode of variabele:

toegangscontrole:

- public
- protected
- <default>
- private

verder:

- abstract
- static
- final

static members

- static "members" (methoden of attributen) zijn members van een klasse, niet van een instantie
- direct te gebruiken vanuit de klasse (geen object nodig)
- geen overerving: static members horen alleen bij de klasse waarin ze zijn gedefinieerd, niet bij subklassen
- subklassen kunnen wel bij de static members van de superklasse, maar krijgen daar geen eigen versie van
- static methoden kunnen niet abstract zijn
- static methoden kunnen alleen bij andere static members

static members - voorbeeld

```
class Dier {
  static String leefgebied = "land";
}
class Vis extends Dier{}
class Vogel extends Dier{}

class TestDier{
  public static void main(String args[])  {
     Vogel tweety = new Vogel();
     tweety.leefgebied = "lucht";
     Vis nemo = new Vis();
     nemo.leefgebied = "water";

     System.out.println(tweety.leefgebied);
        // uitvoer: water
  }
}
```

static imports

- Static members van een klasse kunnen worden geïmporteerd
- Daarna kunnen die members direct worden gebruikt, zonder klassenaam ervoor

Voorbeeld:

```
import static java.lang.Math.*; // ook bij java.lang
                               // is package naam verplicht
import static java.lang.String.valueOf; // overloaded methoden
                              // geen haakjes, geen parameters
class ImportStatic {
  public static void main(String args[]) {
    String wortelVanPi = valueOf(sqrt(PI));
    System.out.println(wortelVanPi);
  }
}
```

Toegangscontrole

- Kan een methode van de ene klasse een member van een andere klasse benaderen?

```
A a = new A();
a.doeIets(); // lukt dit?
```

- Kan een subklasse bij een member van z'n superklasse?

```
class B extends  A {
  public void gebruikSuper(){
     doeIets(); // lukt dit?
  }
}
```

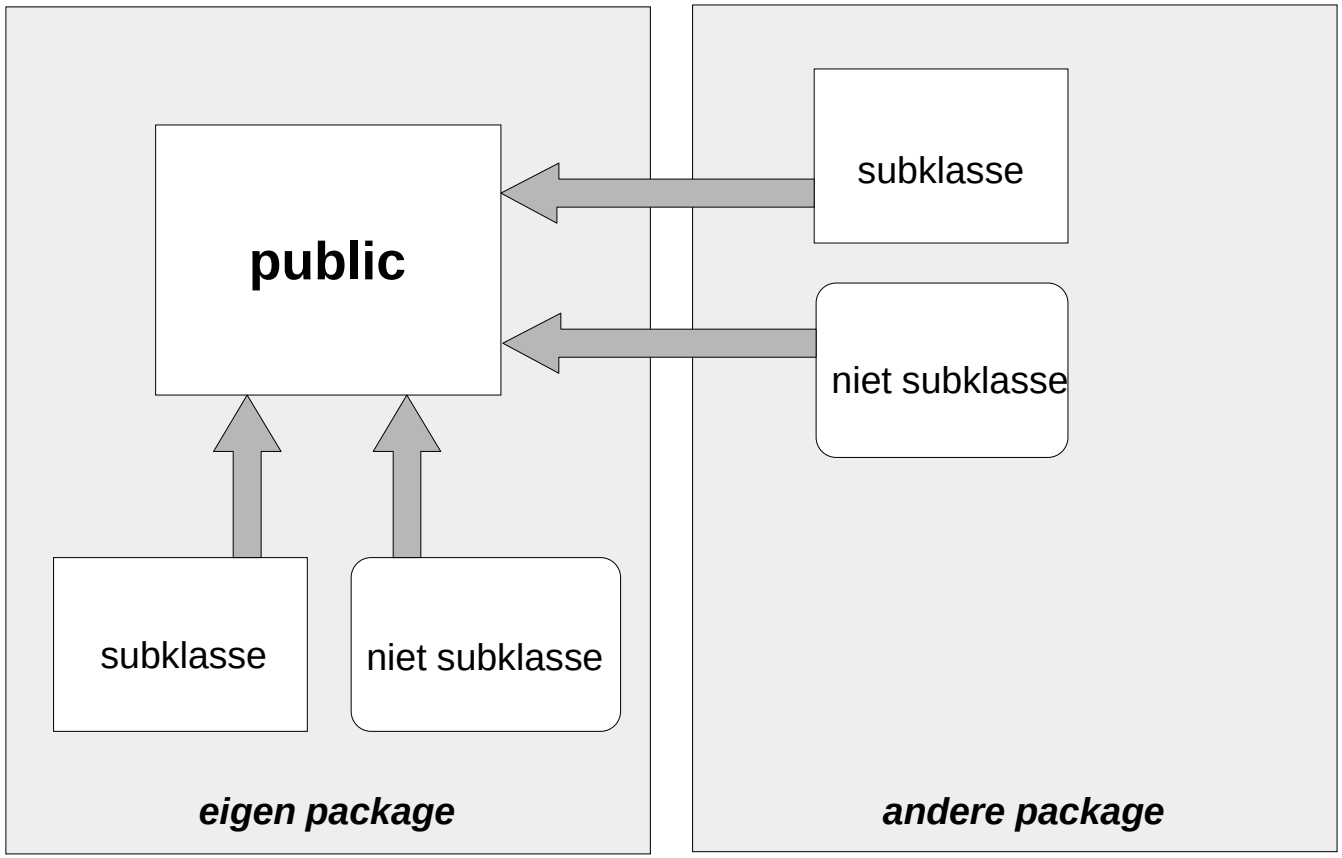
Modifiers voor toegangscontrole

	public	protected	default (zonder modifier)	private
Zelfde klasse	ja	ja	ja	ja
Subklasse in zelfde package	ja	ja	ja	nee
Niet subklasse in zelfde package	ja	ja	ja	nee
Subklasse in andere package	ja	ja, *via overerving*	nee	nee
Niet subklasse in andere package	ja	nee	nee	nee

Een public *class* kan ook benaderd worden vanuit andere packages, zonder public is een klasse alleen binnen de eigen package bereikbaar.

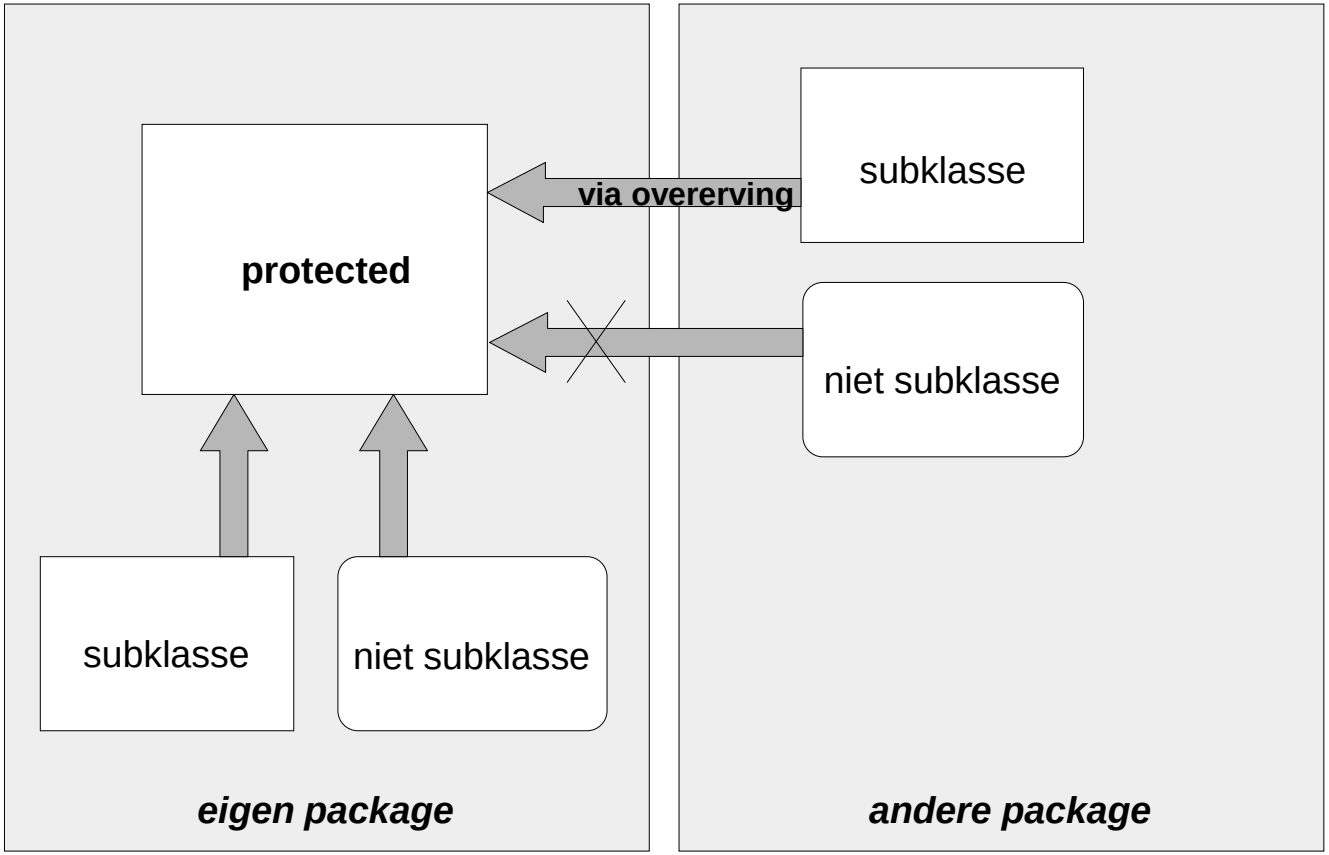
public

Public members zijn benaderbaar vanuit klassen binnen en buiten de eigen package



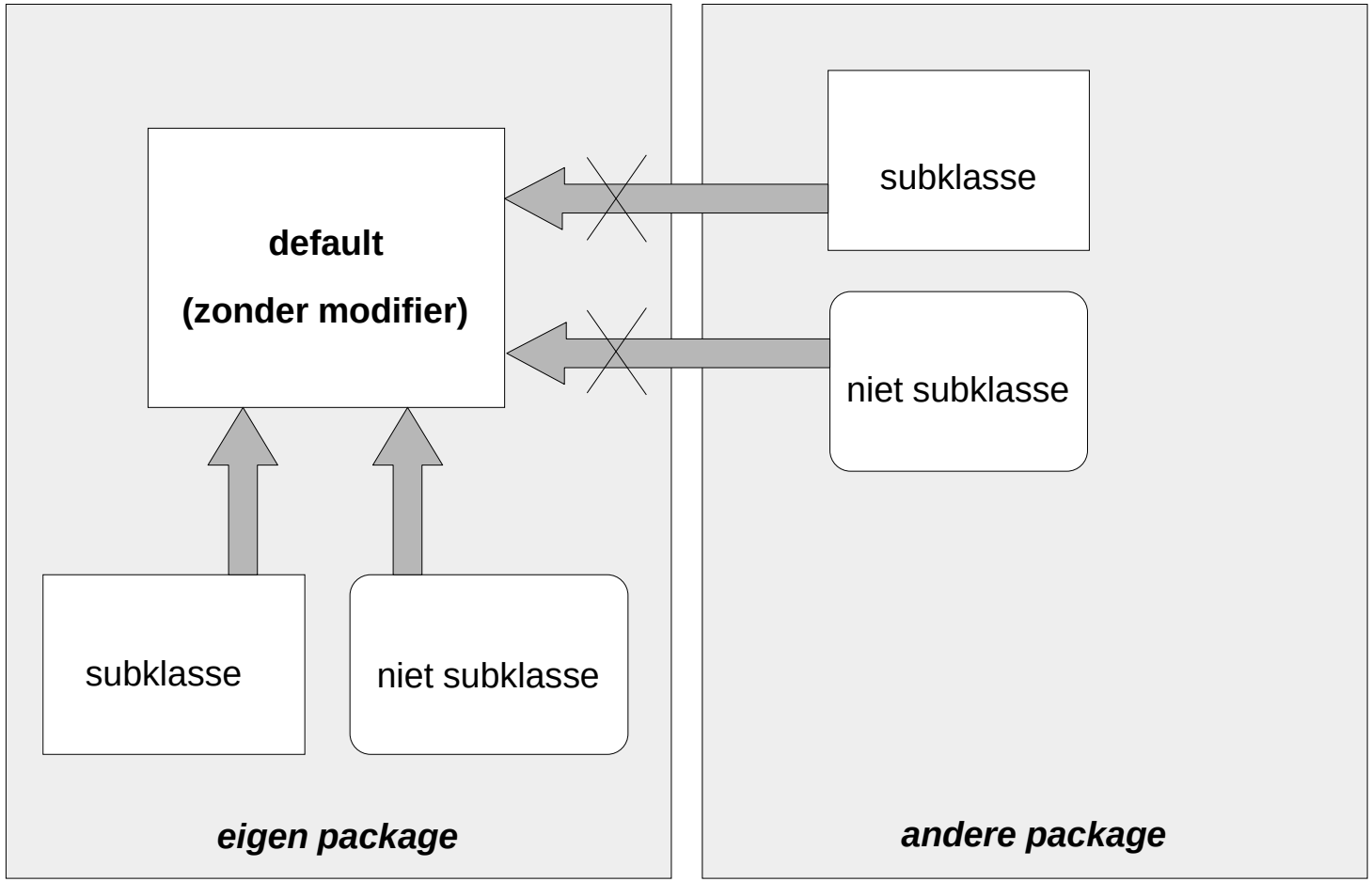
protected

Protected members zijn benaderbaar vanuit alle klassen binnen de eigen package, en *via overerving* vanuit subklassen in andere packages



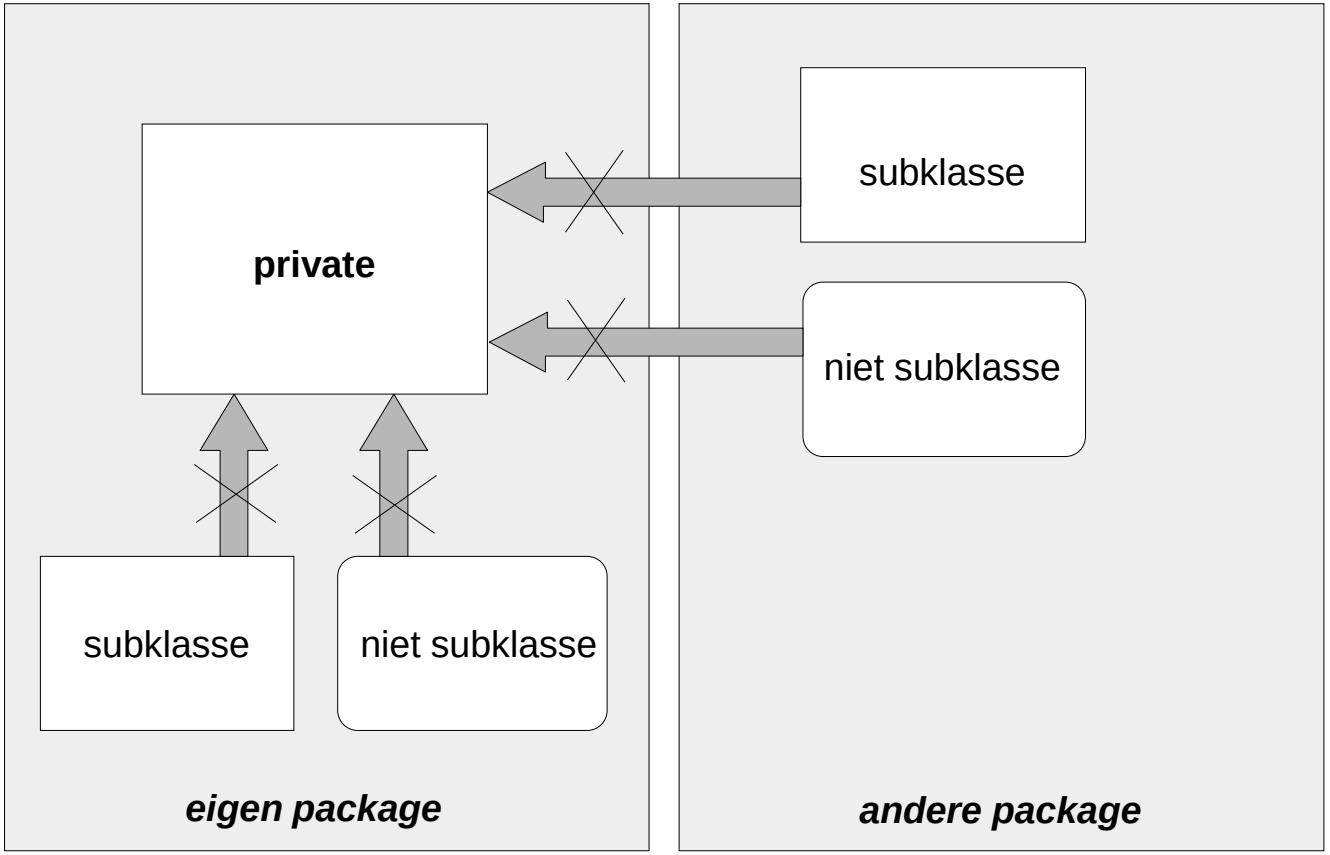
standaard, zonder modifier

Zonder modifier zijn members alleen te bereiken door klassen in dezelfde package



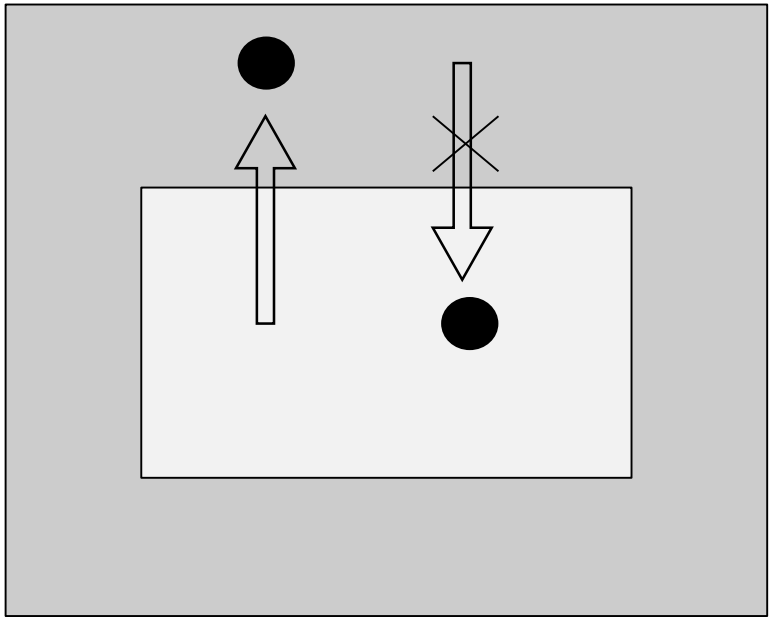
private

private members zijn alleen benaderbaar binnen de eigen klasse.
Toegang tot private attributen wordt via public methoden geregeld.



Lokale variabelen

- Een lokale variabele (binnen een blok en/of binnen een methode) is alleen bereikbaar binnen datzelfde blok of binnen geneste subblokken.
- Lokale variabelen worden ook *automatic* variabelen genoemd: ze worden automatisch verwijderd aan het einde van hun scope.



Overriding en toegangscontrole

- Bij overriding dient de methode in de subklasse dezelfde toegang of bredere toegang te krijgen als de overschreven methode.
- Toegankelijkheid die op niveau van superklasse gegarandeerd is moet op z'n minst ook voor subklasse gelden.

```
class Superklasse {
  protected void mijnMethode(){}
}

class Subklasse extends Superklasse {
  public void mijnMethode(){}
}

class SubklasseFout extends Superklasse {
  void mijnMethode(){}
}
//error: attempting to assign weaker access privileges
```

accessor methoden - getters en setters

```
class Leverancier {
     private int kortingsPercentage = 5;
     private boolean preferredSupplier;
     public int getKortingsPercentage(){
       return kortingsPercentage;
     }
     public void setKortingsPercentage(int percentage){
       if (percentage >=0 && percentage < 100) {
         kortingsPercentage = percentage;
       }
     }
     public boolean isPreferredSupplier (){
       return preferredSupplier;
     }
     public void setPreferredSupplier(boolean aanUit){
       preferredSupplier = aanUit;
     }
}
```

final

- final klasse:
  - hiervan kan geen subklasse worden gemaakt
  - een final klasse kan dus ook niet abstract zijn
- final variabelen:
  - waarde kan na initialisatie niet worden gewijzigd
  - final reference variabele: kan niet naar ander object verwijzen, object zelf kan wel wijzigen
  - final instance variabele: waarde moet direct bij declaratie, in een "instance initializer", of in de constructor worden geïnitialiseerd
  - final static variabele: waarde moet direct bij declaratie, of in een "static initializer" worden geïnitialiseerd (initializers worden in het volgende hoofdstuk behandeld)
- final methoden:
  - kunnen niet worden overschreven in een subklasse
- final parameters van methoden:
  - compiler controleert of parameterwaarden niet worden gewijzigd in de methode

H6: opgaven

H7 - Constructors en overloading

Doelen:

- Constructors kunnen aanmaken en gebruiken
- Overloading kunnen toepassen op constructors en methoden
- Bestaande code kunnen hergebruiken m.b.v. super en this
- Gebruik kunnen maken van geneste klassen

OCA boek: H1 p16 t/m 20
H4 p186, 196 t/m 204

Constructor - inleiding

Een constructor ...

- heeft dezelfde naam als de klasse
- wordt gebruikt om nieuwe objecten aan te maken
- bevat code om instance variabelen te initialiseren
- een klasse zonder constructor heeft een default constructor
- ook een abstracte klasse heeft een constructor

```
class Rechthoek {
  int hoogte, breedte;

 Rechthoek() {
    hoogte = 5;
    breedte = 10;
  }
}
```

static initializers

- blok code met "static" ervoor
- bedoeld om static variabelen te initialiseren
- wordt uitgevoerd als de klasse voor het eerst wordt geladen
  - als een object van de klasse wordt aangemaakt
  - als een static methode van de klasse wordt aangeroepen
- nodig als de waarde van een final static variabele gegenereerd moet worden in een blok code
- anders is het handiger om direct te initialiseren bij het declareren

static initializer - voorbeeld

```
import java.util.*;
import java.text.*;

class StartTijd {
  private final static String nu;

  static {
    Locale nl =new Locale("nl","NL");
    DateFormat df =
          new SimpleDateFormat("EEEE dd MMMM yyyy kk:mm:ss.SSS",nl);
    nu = df.format(new Date());
  }

  public static void main(String args[]) {
    System.out.println(nu);
  }
}
```

Anonieme inner classes (geen examenstof)

- Een **anonieme inner class** is een class zonder naam, waarvan direct een object wordt gemaakt.
- Het object wordt aangemaakt door achter new een implementatie te geven.
- Te gebruiken als de (anonieme) class alleen op die plek wordt gebruikt.
- Lambda expressies (laatste hoofdstuk) zijn hiervan afgeleid

```
abstract class Formatter{
  abstract String format(int x);
  static Formatter getInstance(int low, int high) {
    return new Formatter(){    // object van subclass, met reference type
      @Override                // superclass: dus geen nieuwe methods
      String format (int x) { // toevoegen, alleen bestaande overriden
        int range = high-low;
        double pct=(double)(x - low)/range*100;
        return String.format("%s %%",pct);
      }
    };
  }
}
```

Overloading

- binnen een klasse moet de signatuur van een methode of constructor uniek zijn
- het return type hoort niet bij de signatuur
- methoden die iets retourneren kunnen ook als void worden aangeroepen
- methode overloading:
  - om zelfde soort bewerking toe te kunnen passen op verschillende datatypen (aantal parameters is gelijk, datatype verschilt)
  - om extra opties mee te kunnen geven bij een bewerking (aantal parameters verschilt)
- constructor overloading:
  - om meer of minder instance variabelen te initialiseren
  - default parameterloze constructor bestaat alleen als er geen andere constructors zijn
  - parameterloze constructor toevoegen: wordt impliciet aangeroepen bij het aanmaken van een instantie van een subklasse

methode overloading - voorbeeld

`static double`	**`max`**`(double a, double b)` Returns the greater of two `double` values.
`static float`	**`max`**`(float a, float b)` Returns the greater of two `float` values.
`static int`	**`max`**`(int a, int b)` Returns the greater of two `int` values.
`static long`	**`max`**`(long a, long b)` Returns the greater of two `long` values.

constructor overloading - voorbeeld

`StringBuilder``()`

Constructs a string builder with no characters in it and an initial capacity of 16 characters.

`StringBuilder``(`**`CharSequence`**` seq)`

Constructs a string builder that contains the same characters as the specified `CharSequence`.

`StringBuilder``(int capacity)`

Constructs a string builder with no characters in it and an initial capacity specified by the `capacity` argument.

`StringBuilder``(`**`String`**` str)`

Constructs a string builder initialized to the contents of the specified string.

instance initialization block

- block zonder naam op klasse-niveau
- compiler kopieert de code naar het begin van elke constructor
- de rest van de code in de constructor wordt daarna uitgevoerd

```
class Rechthoek {
  private int breedte, hoogte;
  private String naam;
  { naam = "Rechthoek"; }
  Rechthoek(){breedte=5; hoogte=5;}
  Rechthoek(int b, int h){
   breedte=b;
   hoogte=h;
  }
 }
```

this

this verwijst naar het huidige object

1. om onderscheid te maken tussen parameter en instance variabele
2. om binnen een constructor een andere constructor aan te roepen

```
public class Rechthoek {
  private int hoogte, breedte;
  public Rechthoek(int hoogte, int breedte) {
     this.hoogte = hoogte; // voorbeeld van 1.
     this.breedte = breedte;
  }
  public Rechthoek() {
    this(5,5); // voorbeeld van 2.
  }
  public int getHoogte(){ return hoogte; }
  public void setHoogte(int hoogte) {
    this.hoogte=hoogte;
 }
}
```

H7: opgave 1 en 2

super

super verwijst naar de superklasse

1. om in een overschreven methode gebruik te maken van de code uit de superklasse
2. om in een constructor gebruik te maken van de constructor van de superklasse
3. verwijzen naar een attribuut uit de superklasse

super - voorbeeld override

```
public String toString() {
  String tekst = super.toString();
  int positie = tekst.indexOf("@");
  return "Object van " + tekst.substring(0, positie);
}

// uitvoer in klasse Rechthoek:
// Object van Rechthoek

// beter: gebruik getClass().getName()(inclusief packagenaam)
// of: getClass.getSimpleName() (zonder packagenaam)

public String toString() {
  String tekst = getClass().getSimpleName();
  return "Object van " + tekst;
}
```

super - constructor voorbeeld 1

```
public class OudersEerst {
  public static void main(String args[]){Kind kind = new Kind();}
}
class Grootouder {
  Grootouder(){ System.out.println("Grootouder");}
}
class Ouder extends Grootouder {
  Ouder(){ System.out.println("Ouder");}
}
class Kind extends Ouder {
  Kind(){ System.out.println("Kind"); }
}
// uitvoer: Grootouder
//          Ouder
//          Kind
```

Impliciet wordt in elke constructor eerst super() aangeroepen

super - constructor voorbeeld 2

```
public class OudersEerst {
  public static void main(String args[]){Kind kind = new Kind();}
}
class Grootouder {
  Grootouder(String naam){ System.out.println("Grootouder: "+ naam);}
}
class Ouder extends Grootouder {
  Ouder(){ System.out.println("Ouder"); }
}
class Kind extends Ouder {
  Kind(){ System.out.println("Kind"); }
}
  constructor Grootouder in class Grootouder cannot be applied to given types;
  Ouder(){ System.out.println("Ouder");}
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
```

super - constructor voorbeeld 3

```
public class OudersEerst {
  public static void main(String args[]){Kind kind = new Kind();}
}
class Grootouder {
  Grootouder(String naam){ System.out.println("Grootouder: "+ naam);}
}
class Ouder extends Grootouder {
  Ouder(){ super("Oma"); // expliciete aanroep
  System.out.println("Ouder");}
}
class Kind extends Ouder {
  Kind(){ System.out.println("Kind"); }
}
// uitvoer: Grootouder: Oma
//          Ouder
//          Kind
```

super - constructor voorbeeld 4

```
public class OudersEerst {
  public static void main(String args[]){Kind kind = new Kind();}
}
class Grootouder {
  Grootouder(){System.out.println("Grootouder"); // extra constructor
  Grootouder(String naam){ System.out.println("Grootouder: "+ naam);}
}
class Ouder extends Grootouder {
  Ouder(){System.out.println("Ouder");}
}
class Kind extends Ouder {
  Kind(){ System.out.println("Kind"); }
}
// uitvoer: Grootouder
//          Ouder
//          Kind
```

super en this in constructors

- als eerste aanroep binnen constructor kan super(...) of this(...) worden meegegeven
- zonder aanroep wordt impliciet als eerste regel super() aangeroepen
- als eerst this(...) wordt gegeven, dan wordt impliciet de super() van de aangeroepen constructor gebruikt: de eigen constructor heeft dan dus geen super() aanroep
- code van een instance initializatie blok komt direct na de super()

super - verwijzing naar attribuut

```
class Dier {
  int aantalPoten = 0;
   Dier() {
    aantalPoten = 4;
  }
}
class Spin extends Dier {
  int aantalPoten = 8; // aantalPoten wordt hier opnieuw gedeclareerd
  void toonAantalPoten(){
    System.out.println("aantal poten Spin: " + aantalPoten);
    System.out.println("aantal poten Dier: " + super.aantalPoten);
  }
}
// uitvoer bij aanroep toonAantalPoten():
// aantal poten Spin: 8
// aantal poten Dier: 4
// impliciet is de constructor van Dier aangeroepen: super()
```

Covariant return types

- Een overridden methode in een subklasse moet zelfde signatuur hebben als in de superklasse
- Het returntype moet hetzelfde zijn **of een subtype** van het returntype in de superklasse

```
class SuperKlasse {
  SuperKlasse getElement() {
    return this;
  }
}

class SubKlasse extends SuperKlasse {
  SubKlasse getElement() {
    return this;
  }
}
```

toegang tot constructoren

- public: alle klassen binnen of buiten de package kunnen een instantie aanmaken
- default: voor klassen binnen eigen package. Gaat meestal om niet-public hulpklassen, die door andere public klassen worden gebruikt
- protected: voor alle klassen binnen de package, en voor alle subklassen. Meestal gebruikt voor abstracte klassen, zodat subklassen impliciet super() aan kunnen roepen.
- private: als aanmaken van instanties gecontroleerd moet gebeuren. Voorbeeld: singleton pattern, of final hulpklasse met alleen static methoden
- De default constructor krijgt dezelfde toegang als de betreffende klasse
- Een enum heeft impliciet een private default constructor

private constructor - singleton pattern

```
public class Singleton {
  private static Singleton instance;
  private Singleton(){}
  public static Singleton getInstance(){
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

H7: opgave 3

H8 - Conversie, polymorfisme en interfaces

Doelen:

- Begrijpen wat type casting is en dit kunnen toepassen
- Gebruik kunnen maken van polymorfisme
- Het datatype van een object kunnen bepalen met instanceof (niet meer in OCA examen)
- Het nut van interfaces begrijpen

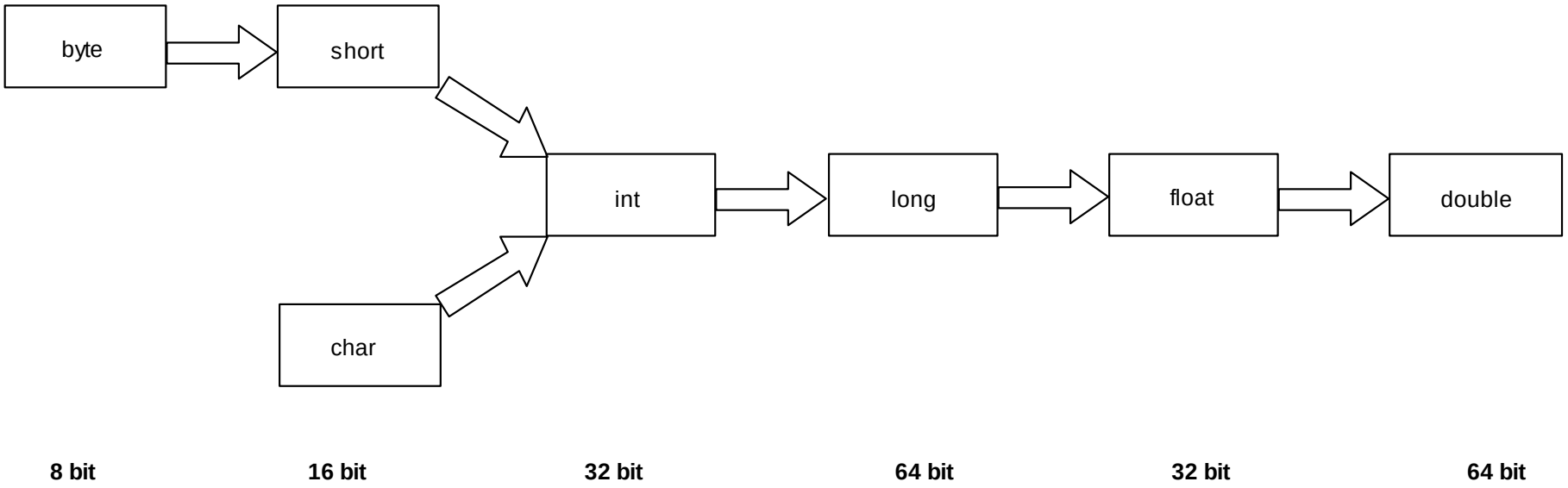
OCA boek H5 p279 t/m 288 (polymorfisme)
p266 t/m 279 (interfaces)

automatische type conversie

Automatische type conversie van primitieve typen:

- als Java expressies evalueert
- als Java waarden toekent
- als Java methoden aanroept en parameterwaarden doorgeeft

Geen verlies van precisie of van informatie bij "verbredende" conversie:



Type casting

- Nodig bij "versmallende" conversie
- Hierbij treedt dan verlies aan precisie op
- syntax: **`(nieuw_type)expressie`**

Voorbeeld:

```
int uitkomst = (int) (2.5 + 3.4);
// uitkomst is nu 5
```

- null kan worden gecast naar een type dat null kan zijn

```
String a = (String)null; // compileert
int x = (int)null; // foutmelding: inconvertable types
                   // required: int
                   // found <null>
```

Automatische type promotie

Het resultaat van een expressie met operanden van type t kan buiten de grenzen van type t vallen:

bijvoorbeeld

```
byte a = 100, b = 100;
int  c = a * b;  // moet in int, 10.000 past niet in byte
```

Automatische type promotion in expressies:

- als een van beide getallen een double is, wordt het andere getal naar double gecast
- anders, als een van beide getallen een float is, wordt het andere getal een float
- anders, als een van beide getallen een long is, wordt het andere getal een long
- anders worden beide getallen een int

`byte d = a - b;` // foutmelding bij compileren:

```
possible loss of precission
required: byte
found: int
```

H8: opgave 1

polymorfisme

- Afgeleid van Grieks voor "meerdere vormen"
- Een variabele kan als volgt worden aangemaakt:

SuperType a = new SubType();

- Variabele a heeft nu eigenschappen van SuperType en van SubType
  - **Reference type** = SuperType;
  - **Object type** = SubType;

- **Reference type** bepaalt ***wat*** je met de variabele kunt doen (compiler controleert daarop)

- **Object type** bepaalt ***hoe*** het wordt uitgevoerd (alleen bij instance methods)

polymorfisme - dynamic binding

```
abstract class Boodschap {
  void verzenden()  { System.out.println("boodschap verzonden"); }
}

class Email extends Boodschap{
  String antwoordAdres = "info@5hart.nl";
  void verzenden()  { System.out.println("email verzonden"); }
}

class BoodschapApp {
  public static void main(String args[])  {
    Boodschap b = new Email();
    b.verzenden(); // dynamic binding: alleen voor instance methoden
  }                // bij static methoden pakt ie de methode
}                  // van het gedeclareerde type (hier: Boodschap)

// uitvoer: email verzonden
```

polymorfisme - downcasting

```
abstract class Boodschap {
  void verzenden()  { System.out.println("boodschap verzonden"); }
}

class Email extends Boodschap{
  String antwoordAdres = "info@5hart.nl";
  void verzenden()  { System.out.println("email verzonden"); }
}

class BoodschapApp {
  public static void main(String args[])  {
    Boodschap b = new Email();
    // b.antwoordAdres is niet bekend, dit kan:
    System.out.println(((Email)b).antwoordAdres);
    // in dit geval was dit beter geweest: Email e = new Email();
  }
}
// uitvoer: info@5hart.nl
```

polymorfisme - downcasting en instanceof

- downcasting kan naar het type waarmee het object is aangemaakt, of naar een supertype daarvan
- anders runtime ClassCastException
- controle kan met instanceof:

```
if(b instanceof Email) {
  Email mail = (Email)b;
  System.out.println(mail.antwoordAdres);
}
```

- instanceof geeft true als object van opgegeven type is
- als b niet null is geeft instanceof Object dus altijd true
- als b wel null is geeft instanceof altijd false,
- al kan null naar elk referentie type worden gecast

polymorfisme - vergelijking met == of !=

```
class Boodschap{}
class Email extends Boodschap{}
class Brief extends Boodschap{}

Email email = new Email();
Brief brief = new Brief();
Boodschap boodschap = email;

System.out.println(boodschap==email); // compileert, is true
System.out.println(boodschap==brief); // compileert, is false

System.out.println(email==brief); // compileert niet
// incomparable types: Email and Brief
```

polymorfisme - instanceof

```
class Boodschap{}
class Email extends Boodschap{}
class Brief extends Boodschap{}

Email email = new Email();
Brief brief = new Brief();
Boodschap boodschap = email;

System.out.println(boodschap instanceof Email);
                                                    // compileert, is true
System.out.println(boodschap instanceof Brief);
                                                    // compileert, is false

System.out.println(email instanceof Brief); // compileert niet
// incomparable types: Email can not be converted to Brief
```

polymorfisme - getClass()

- Wat is het type waarmee een object is aangemaakt?
- De methode getClass() van Object geeft een Class object terug
- Class heeft o.a. de methoden getName() en getSimpleName()

```
package com.vijfhart.cursus;
abstract class Boodschap{}
class Email extends Boodschap{}
class Test {
  public static void main (String args[]){
    Boodschap boodschap = new Email();
    System.out.println(boodschap.getClass().getName());
    System.out.println(boodschap.getClass().getSimpleName());
  }
} // uitvoer:
  // com.vijfhart.cursus.Email
  // Email
```

de methode equals() van Object

- equals() is bedoeld om te worden overschreven
- objecten hebben identity (plek in het geheugen) en state (actuele data van het object)
- standaard geeft equals() true als twee objecten dezelfde identity hebben
- equals() wordt meestal overschreven om te testen of objecten dezelfde state hebben

equals() - regels van gelijkwaardigheid

- reflexief: voor elk object x geldt dat x.equals(x) is true
- symmetrisch: als x.equals(y), dan ook y.equals(x)
- transitief: als x.equals(y) en y.equals(z), dan ook x.equals(z)
- consistent: bij ongewijzigde data, blijft x.equals(y) hetzelfde
- als referentie naar object a niet null is, dan is a.equals(null) false

```
class Punt{
  int x, y;
  Punt(int x, int y){ this.x=x; this.y=y; }
  @Override
  public boolean equals(Object obj) {
    if(obj instanceof Punt){
      Punt other = (Punt)obj;
      if(this.x==other.x && this.y==other.y){ return true; }
    }
    return false;
  }
}
```

H8: opgave 2 en 3

interfaces

- een klasse heeft één superklasse, maar kan meer interfaces implementeren
- interface lijkt op abstracte klasse: alleen abstracte methoden en/of variabelen
- een interface kan meerdere super-interfaces hebben
- interfaces kunnen ook default methoden en static methoden hebben (sinds Java 8)
- methoden zijn impliciet public, variabelen zijn impliciet public final static (constanten)
- subklasse heeft *is een* relatie met zowel superklasse als alle interfaces

Voorbeeld - een interface voor een applicatie die een bon uitprint voor een klant:

```
interface Verkoopbaar {
  String naam();
  float prijs();
}
```

interfaces - voorbeeld implementatie

```
class Artikel implements Verkoopbaar {
  private String type;
  private String merk;
  private double prijs;
 [... Artikel-specifieke code ...]
  @Override
  public String naam(){   // implementatie van naam() in Verkoopbaar
    return type+" "+ merk;
  }

  @Override
  public float prijs(){   // implementatie van prijs() in Verkoopbaar
    return (float)prijs;
  }
}
class Cursus implements Verkoopbaar, Inplanbaar, Schrijfbaar {...}
```

interfaces - voorbeeld gebruik

```
class PrintBonApp {
  public static void main (String args[]){
    Verkoopbaar vkb1 = new Cursus("Breien", 1000);
    Verkoopbaar vkb2 = new Artikel("Breipen", 2.55);
    printBon(vkb1);
    printBon(vkb2);
  }

   static void printBon(Verkoopbaar v){
      String naam = v.naam();
      float prijs = v.prijs();
      System.out.println(naam+ " kost " + prijs);
   }
}
```

interfaces - voordelen

- loose coupling: alleen publieke methoden worden aangeroepen, implementatie geheel verborgen
- verkort ontwerptijd vd applicatie: methoden en gewenste interacties van interfaces kunnen als "contracten" naar ontwikkelaars worden gestuurd
- ontwikkelaars hoeven de implementaties van andere interfaces niet te kennen
- extra flexibiliteit: later kunnen andere implementerende klassen worden gebruikt, andere code hoeft dan niet te worden aangepast

interfaces - overerving

```
interface Startable {
  void start();
}

interface Stoppable extends Startable {  // uitbreiding op Startable
  void stop();
}

class Auto implements Stoppable { //= implements Stoppable, Startable
   public void start(){
      System.out.println("auto start")
   }

   public void stop(){
      System.out.println("auto stopt")
   }
}
```

interfaces - geen constructor

```
interface Klant {
  void betalen(String product);
}

Klant k1 = new Persoon(); // dit kan als Persoon Klant implementeert
Klant k2 = new Bedrijf(); // dit kan als Bedrijf Klant implementeert
Klant k3 = new Klant();   // dit kan niet!
```

Java API interfaces - CharSequence

CharSequence wordt door o.a. String en StringBuilder geïmplementeerd.

`int length()`	bedoeld om het aantal karakters in de sequence te retourneren
`char charAt(int positie)`	bedoeld om het karakter op de opgegeven positie te retourneren
`CharSequence subSequence( int start, int einde)`	moet de CharSequence retourneren tussen de opgegeven begin- en eindpositie.
`String toString()`	bedoeld om de CharSequence als String te retourneren, met de letters in de zelfde volgorde.

voorbeeld gebruik CharSequence

```
class Tekst {
  private CharSequence waarde;
  public CharSequence append(CharSequence cs){
    StringBuilder sb = new StringBuilder(waarde);
    // StringBuilder heeft constructor waar CharSequence in kan
    // ... en append waar CharSequence in kan
    return sb.append(cs);
    // append retourneert een StringBuilder
    // en dat is zelf ook een CharSequence
  }
}
```

Appendable

Appendable maakt gebruik van CharSequence: interfaces die interacteren met andere interfaces maken ze breed toepasbaar.

`Appendable append(char c)`	voegt een character toe aan de Appendable
`Appendable append(CharSequence cs)`	voegt de opgegeven CharSequence toe aan de Appendable
`Appendable append(CharSequence cs, int start, int einde)`	voegt het opgegeven deel van de CharSequence toe aan de Appendable

default methoden (Java 8)

- voor Java 8 konden interfaces niet worden uitgebreid
  - nieuwe methode betekent uitbreiding contract
  - klassen die voorheen de interface implementeerden zouden daarna niet meer gecompileerd kunnen worden
- oplossing: default methoden toevoegen
- deze zijn niet abstract:
  - bestaande implementerende klassen kunnen nog steeds gecompileerd worden
  - ze kunnen abstracte eigen methoden aanroepen
  - ze kunnen overschreven worden in implementerende klassen, of
  - ze kunnen de default implementatie in de interface gebruiken
- default methoden mogen geen Object methoden overschrijven

default methoden - voorbeeld

```
// oude interface
interface Verkoopbaar {
   double getPrijs();
   String getNaam();
}

// nieuwe interface
interface Displayable {
  String display();
}

// oude interface krijgt extra functionaliteit, bestaande implementerende klassen
// kunnen nog steeds worden gecompileerd

interface Verkoopbaar extends Displayable {
  double prijs();
  String naam();
  default String display(){
    return String.format("%s: %s",naam(), prijs());
  }
}
```

default methoden - overerving

```
interface I {
   default void doeIets(){ System.out.println("in I");}
}

class A {
  public void doeIets(){ System.out.println("in A"); }
}

class B extends A implements I {
  public static void main (String args[]){
     new B().doeIets(); // uitvoer: in A
  }
}
```

bij overerving methode uit klasse en interface: klasse wint

default methoden - zelfde in meerdere interfaces

```
interface Beweging {
  default String draaien(){
     return "draait om zijn as";
  }
}
interface Applicatie {
  default String draaien(){
    return "staat aan";
  }
}
class Klok implements Beweging, Applicatie {
   public String draaien(){ // override noodzakelijk
     return Applicatie.super.draaien()+": "+Beweging.super.draaien();
   }
}
//  anders bij compilatie:
//  error: class Klok inherits unrelated defaults for draaien()
//  from types Beweging and Applicatie
```

static methoden (Java 8)

- hulpmethoden om met interfaces te werken
- voorheen in losse klassen (zoals Collections (klasse), die met Collection (interface) objecten bewerkt).
- static hulpmethoden mogen nu in de interface zelf geplaatst worden

Voorbeeld:

Een Range interface, geeft een hoogste en een laagste waarde. De default methode pctInRange geeft een opgegeven waarde als percentage terug.

Een static methode of(int x, int y) maakt een Range gebaseerd op de waarden x en y (factory method).

static method voorbeeld - Range

```
interface Range {
   int getLowerBound();
   int getUpperBound();
   default int pctInRange(int value){
     int range = getUpperBound()-getLowerBound();
     double pct = (double) (value - getLowerBound())/range * 100;
     return (int)pct;
   }
   static Range of (int x, int y){
     return new Range(){ // anonieme inner class
        public int getLowerBound(){ return x<y?x:y; }
        public int getUpperBound(){ return x>y?x:y; }
     };
   }
}
// gebruik:
     Range range = Range.of(0, 10);
     System.out.println(range.pctInRange(4));

// uitvoer: 40
```

H8: opgave 4

H9 - Arrays

Doelen:

- Arrays kunnen aanmaken
- Arrays kunnen initialiseren
- Elementen kunnen ophalen of wijzigen via de index
- Kunnen doorlopen van een array met een standard for loop
- Kunnen doorlopen van een array met een enhanced for loop
- De hulpklasse java.util.Arrays kunnen gebruiken

OCA boek H3 p119 t/m 129

Arrays declareren

- Een array is een geïndexeerde groep variabelen van gelijk datatype met gemeenschappelijke naam.
- In Java is een array 0-based (begint met index 0)

Declareren:

```
String[] cursistNamen;
```

of

```
String cursistNamen[];
```

```
String cursistNamen[], docentNaam, lokaalNaam;
String[] cursistNamen, docentNamen, lokaalNamen;
```

Arrays initialiseren

```
int[] nummers;
nummers = new int[10]; // vaste lengte opgeven
of:
int[] nummers = new int[10];
of
int[] nummers = new int[] { 1, 3, 5, 7, 9 ,11, 13, 15, 17, 19 };
of
int[] nummers =  { 21, 23 , 25, 27, 29 , 31, 33, 35, 37, 39 };

// op deze manier vullen mag alleen tijdens het initialiseren,
// dus dit mag niet:

int[] nummers = new int[10];
nummers = { 1, 3, 5, 7, 9 ,11, 13, 15, 17, 19 }; // werkt niet!
nummers = new int[]{ 1, 3, 5, 7, 9 ,11, 13, 15, 17, 19 };//werkt wel!
```

Arrays vullen

```
nummers[0] = 1;
nummers[1] = 3;
nummers[2] = 5;
...

of

for (int i=0; i<nummers.length; i++) {
  nummers[i]=(i+1)*2-1;
}

// geen i<=nummers.length, anders ArrayIndexOutOfBoundsException
```

Pas op voor NullPointerException

```
class NullApp {
  public static void main(String args[]) {
    String[] namen = new String[5];
    namen[0] = "Madelief";
    namen[1] = "Roos";
    namen[2] = "Marjolein";
    namen[4] = "Paardenbloem";
    for(int i=0;i<namen.length;i++) {
      namen[i]=namen[i].toUpperCase();
      System.out.println(namen[i]);
    }
  }
}
java NullApp
MADELIEF
ROOS
MARJOLEIN
Exception in thread "main" java.lang.NullPointerException
```

Voorkomen van NullPointerException

```
for(int i=0;i<namen.length;i++)
    {
      if(namen[i]!=null)
      {
        namen[i]=namen[i].toUpperCase();
        System.out.println(namen[i]);
      }
    }
```

Multidimensionale arrays

```
int[][] tafels = new int [12][10];

for (int i=0;i<tafels.length;i++) {
    // tafels.length == 12
  for (int j=0;j<tafels[i].length;j++) {
    // tafels[i].length == 10
    tafels[i][j]=(i+1)*(j+1);
  }
}
```

index	j →	0	1	2	3	4	5	6	7	8	9
i ↓											
0		1	2	3	4	5	6	7	8	9	10
1		2	4	6	8	10	12	14	16	18	20
2		3	6	9	12	15	18	21	24	27	30
…		…	…	…	…	…	…	…	…	…	…
11		12	24	36	48	60	72	84	96	108	120

Multidimensionale arrays initialiseren

- lengtes van sub-arrays hoeven niet gelijk te zijn
- lengtes van sub-arrays hoeven niet bekend te zijn

Voorbeelden:

```
int[][][] a = new int[4][][]; // correct
int[][][] b = new int[][4][]; // incorrect
int[][][] c = {{{1},{2,3},{4,5,6}},{{7,8}}};
```

H9: opgave 2

de Arrays klasse

Methode	Omschrijving
`static void sort( double[] a)`	Sorteert de double array in oplopende volgorde.
`static int binarySearch( double[] a, double key)`	Geeft aan op welke positie de opgegeven key voorkomt in array a. De array "moet" eerst zijn gesorteerd met Arrays.sort(). Als de methode een negatieve waarde retourneert, is de key niet gevonden. Als de array dubbele waarden bevat, is er geen garantie welke zal worden gevonden.
`static double[] copyOf( double[] original, int length)`	Maakt een kopie van de originele array, ingekort tot lengte length, of aangevuld met default waarden tot lengte length.
`static double[] copyOfRange( double[] original, int from, int to)`	Maakt een kopie van de originele array, van de opgegeven begin index tot de opgegeven eind index. Als de opgegeven eind-index groter is dan de originele lengte, wordt de kopie aangevuld met default waarden.

Arrays klasse - vervolg

Methode	Omschrijving
`static void fill( double[], double val)`	Geeft alle elementen van de array dezelfde opgegeven waarde.
`static boolean equals( double[] a, double[] b)`	Geeft true als a dezelfde waarden in dezelfde volgorde bevat als b, of als a en b beide null zijn.
`static String toString( double[] a)`	Geeft een string representatie van de array: alle double waarden worden gescheiden door komma's, omgeven door vierkante haken.

Hiernaast zijn er in Java 8 enkele nieuwe Arrays methoden toegevoegd:

- ten behoeve van parallelle verwerking (parallelPrefix(), parallelSet() en parallelSort())
- om de array als Stream te verwerken

Arrays klasse - voorbeeld

```
import java.util.Arrays;
public class SorteerApp {
  public static void main(String args[]) {
    String[] invoer = Arrays.copyOf(args,args.length);
    Arrays.sort(invoer);
    System.out.println(Arrays.toString(invoer));
  }
}
```

uitvoeren: `java SorteerApp dit is een test`

resultaat: `[dit, een, is, test]`

System.arraycopy()

```
int a[] = {1,2,3,4,5,6,7,8,9,10,11};
int[] b = new int[12];
Arrays.fill(b,0); // niet nodig: in array van primitief datatype
                  // krijgt elk element een default waarde

// kopieer nu van array a - 4e element
//             naar b - 3e element,
//             7 elementen lang:
System.arraycopy(a,3,b,2,7);
System.out.println(Arrays.toString(b));

// uitvoer: [0, 0, 4, 5, 6, 7, 8, 9, 10, 0, 0, 0]
```

Enhanced for loop

```
// gewone for loop:
for (int i=0; i<args.length; i++){
  // statements;
}

// enhanced for loop:
for (String waarde: args){
  //statements;
}
```

- ook bekend als *for each loop* of *for in loop*
- geen gevaar voor ArrayIndexOutOfBoundsException

Enhanced for loop - wijzigingen binnen loop

```
int[] getallen = {1, 2, 3, 4, 5 };
for (int x : getallen){    // "pass by value"
   x++;
}
System.out.println(Arrays.toString(getallen));
// uitvoer: [1, 2, 3, 4, 5]

StringBuilder[] woorden =
             {new StringBuilder("zes"), new StringBuilder("zeven")};
for (StringBuilder woord : woorden ){  // "pass by reference"
  woord.append("entwintig");
}
System.out.println(Arrays.toString(woorden));
// uitvoer: [zesentwintig, zevenentwintig]
```

Varargs - variabel aantal argumenten

Het is mogelijk om een variabel aantal argumenten van een datatype mee te geven

Voorbeeld:

```
void toonNamen(String ... namen) {
  // code
}
```

Bij het compileren wordt namen dan omgezet in een array.

Aanroep:

```
toonNamen("Kwik","Kwek","Kwak");
```

Varargs - voorbeeld

```
class VariableArgs {

    public static void toonNamen(String... namen)  {
      for (String naam: namen){
           System.out.println(naam);
      }
    }

    public static void main( String args[] ) {
      toonNamen("Kwik","Kwek","Kwak");
      toonNamen(args);
    }
}
```

Varargs - overwegingen

- een vararg argument is vrijwel gelijkwaardig aan een array argument

Dit mag ook:

```
public static void main (String ... args){}
```

Maar:

- een varargs argument kan alleen als laatste argument aan een methode worden meegegeven
- er mag maar één varargs argument aan een methoden worden meegegeven
- een methode met array argument kan niet worden overschreven door een methode met vararg argument en andersom
- een methode met array argument kan alleen worden aangeroepen met een array, niet met losse element-waarden

Varargs en overloading

```
  private static void a(int... x){
    System.out.println("int... x");
  }
  private static void a(int x, int y) {
    System.out.println("int x, int y");
  }
  private static void a(int x, int y, int z) {
    System.out.println("int x, int y, int z");
  }
  public static void main(String args[]) {
    a(2); // kiest varargs, omdat het niet anders kan
    a(2, 3);
    a(2, 3, 4);
  }
// uitvoer: int... x
            int x, int y
            int x, int y, int z
```

H9: opgave 1 (en 3)

H10 - Collecties

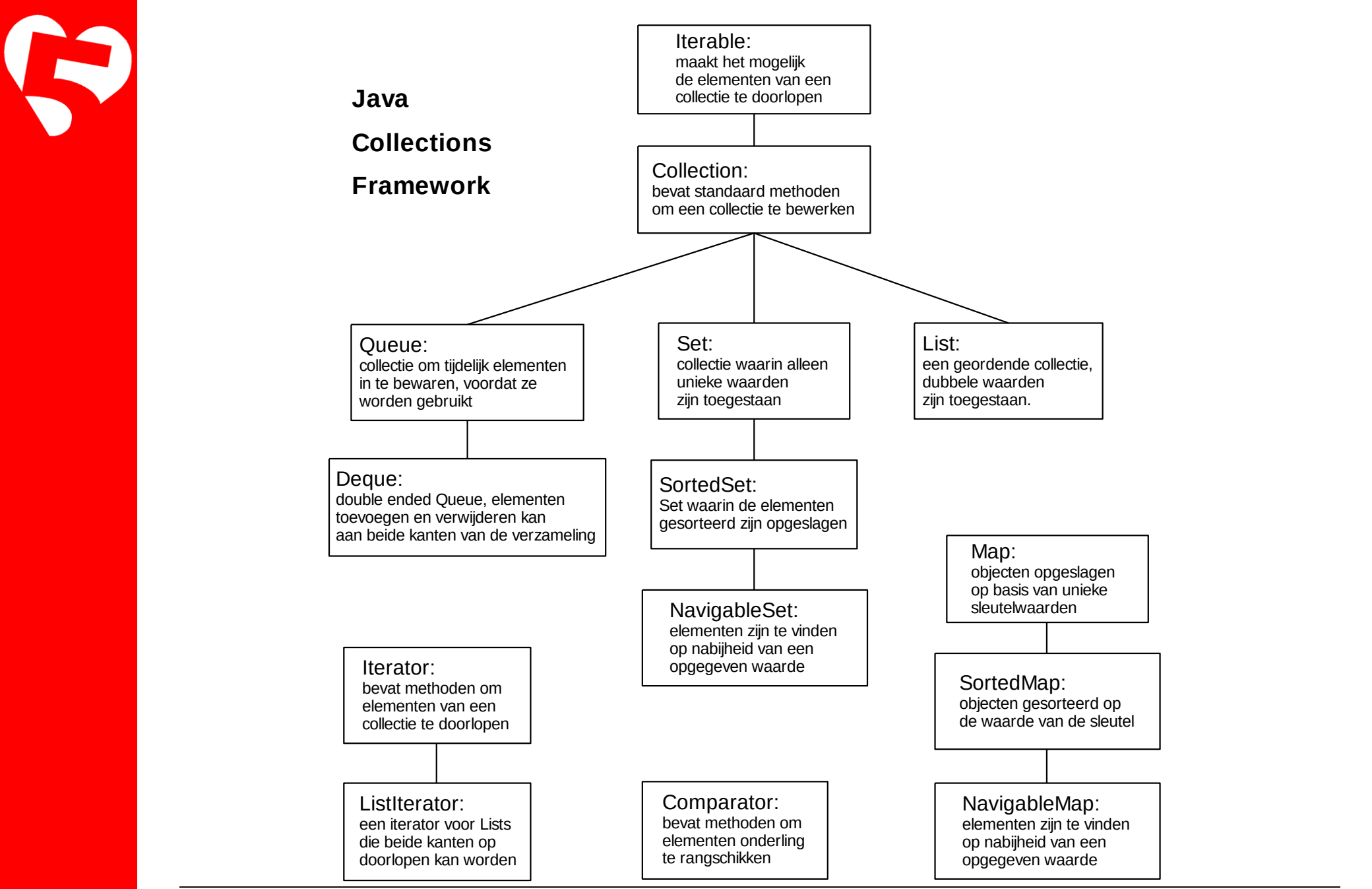
Doelen:

- Verschillen kennen tussen arrays en collecties
- Gebruik kunnen maken van de interface List en de klasse ArrayList uit het Collections framework
- Generics kunnen gebruiken voor het beperken van datatypes in collecties

OCA boek p129 t/m 138

Collecties versus arrays

Arrays	Collecties
vaste lengte	variabele lengte
bevatten elementen van zelfde datatype	kunnen elementen van verschillende datatypen bevatten
elementen zijn alleen te benaderen via index	elementen zijn te benaderen via methoden
bevatten primitieve waarden of objecten	kunnen alleen objecten bevatten


Java
Collections
Framework
Iterable:
maakt het mogelijk
de elementen van een
collectie te doorlopen
Collection:
bevat standaard methoden
om een collectie te bewerken
Queue:
collectie om tijdelijk elementen
in te bewaren, voordat ze
worden gebruikt
Set:
collectie waarin alleen
unieke waarden
zijn toegestaan
List:
een geordende collectie,
dubbele waarden
zijn toegestaan.
Deque:
double ended Queue, elementen
toevoegen en verwijderen kan
aan beide kanten van de verzameling
SortedSet:
Set waarin de elementen
gesorteerd zijn opgeslagen
NavigableSet:
elementen zijn te vinden
op nabijheid van een
opgegeven waarde
Map:
objecten opgeslagen
op basis van unieke
sleutelwaarden
Iterator:
bevat methoden om
elementen van een
collectie te doorlopen
SortedMap:
objecten gesorteerd op
de waarde van de sleutel
ListIterator:
een iterator voor Lists
die beide kanten op
doorlopen kan worden
Comparator:
bevat methoden om
elementen onderling
te rangschikken
NavigableMap:
elementen zijn te vinden
op nabijheid van een
opgegeven waarde

Collections framework

- collecties gebaseerd op **key-value** paren
- collecties waarbij **dubbele waarden** wel of niet zijn toegestaan
- **gesorteerde** en **ongesorteerde** collecties
- verschillende van ophalen van elementen:
  - FIFO (first in first out)
  - LIFO (last in first out)
  - met get() en index
  - via iterator

Implementerende klassen

Naam van implementerende klasse begint vaak met wijze van organisatie (Tree, Array, Linked, Hash) en eindigt dan op de geïmplementeerde interface (List, Deque, Set, Map)

Andere klassen:

Vector: synchronized, thread safe implementatie van List, maakt intern gebruik van een array om elementen in op te slaan

Stack: Een subklasse van Vector met "last-in, first-out" (LIFO) functionaliteit. Na het toevoegen van elementen worden de elementen er in omgekeerde volgorde weer uitgehaald.

Implementerende klassen - vervolg

Klasse	beschrijving
ArrayList ArrayDeque	Intern maakt de collectie gebruik van een array om de elementen op te slaan. Een ArrayList lijkt sterk op een Vector. Een ArrayDeque (double ended queue, spreek uit: "dek") biedt methoden om zowel elementen aan het begin van de queue als aan het eind van de queue te benaderen.
TreeSet TreeMap	De elementen zijn in een boomstructuur opgeslagen, efficiënt voor het zoeken van elementen. TreeSets worden vaak gebruikt als elementen gesorteerd moeten worden opgeslagen.
LinkedList LinkedHashSet LinkedHashMap LinkedBlockingQueue	De volgorde van elementen wordt bepaald door links tussen het ene element en het volgende element. Deze structuur maakt het geschikt voor toepassingen waarin veel wijzigingen plaatsvinden: als een element wordt toegevoegd, verwijderd of verplaatst, hoeven alleen de links te veranderen.
Hashtable HashSet HashMap LinkedHashSet LinkedHashMap	Elementen worden gevonden via een hashCode() implementatie.

Eigenschappen van de interfaces

interface	beschrijving	methoden
Iterable	Collecties die Iterable implementeren kunnen met een enhanced for loop worden doorlopen.	Met behulp van de methode **iterator()** wordt een iterator aangemaakt. De elementen uit de verzameling kunnen vervolgens met een loop worden doorlopen. Deze iterator wordt impliciet gebruikt bij een enhanced for loop.
Collection	De **basisinterface** van verschillende soorten verzamelingen elementen. Wordt gebruikt als alleen de basiseigenschappen van de verzameling van belang zijn. Het is de bedoeling dat implementaties een parameterloze constructor krijgen, en een constructor waaraan een collectie elementen kan worden meegegeven.	Methoden om elementen toe te voegen, te verwijderen, te checken of ze aanwezig of leeg zijn, om de collectie in een Array om te zetten, om de lengte op te vragen: add(), remove(), clear(), contains(), toArray(), isEmpty(), size(). Aan sommige methoden kun je een collectie elementen meegeven om te verwerken (addAll(), removeAll(), containsAll(), retainAll()).
List	Een collectie waarbij **de positie van de elementen** bij het invoeren en bij het opvragen **kan worden bepaald**. Dubbele waarden zijn toegestaan, lege waarden ook.	Onder meer overloaded methoden waaraan de positie kan worden meegegeven, zoals: add() en remove(). Andere methoden om elementen op te halen of van waarde te veranderen, of de positie te bepalen: get(), set(), indexOf(), lastIndexOf(). Met subList() kan een List worden gemaakt van elementen tussen twee posities in.

Eigenschappen van de interfaces - vervolg

interface	beschrijving	methoden
Set	Een collectie waarin **elke waarde maar één keer mag voorkomen**. Er mogen dus geen elementen e1 en e2 in voorkomen waarvoor geldt e1.equals(e2). De waarde null mag in sommige implementaties niet voorkomen, in andere wel, maar dan ook maar één keer.	Dezelfde methoden als van Collection, alleen enkele hebben extra restricties. Zo mag add() geen elementen toevoegen die al bestaan.
Queue	Een collectie waarin **tijdelijk elementen worden bewaard, om later te bewerken**. De queue kan gevuld en geleegd worden. Elementen toevoegen kan zolang de queue niet vol is, en elementen eruit halen om te verwerken kan zolang de queue niet leeg is.	Methoden om elementen toe te voegen, uit de queue te halen, of om ze te bekijken. Als een volle of lege queue een uitzondering is, kunnen methoden gebruikt worden die een exception opleveren als de aanroep mislukt (**add(), remove() en element()**). Als een volle of lege queue geen uitzondering is, gebruik dan de alternatieve methoden **offer(), poll() en peek()**. offer() retourneert false() als de queue vol is, poll() en peek() geven null als de queue leeg is.

Oude notatie

```
import java.util.*;

class CollectionApp {
   public static void main(String args[])  {
    Collection c = new ArrayList(); // datatype onbekend, onwenselijk
    c.add("Collection");
    c.add("array");
    c.add("Set");
    c.add("array");
    Iterator i = c.iterator(); // geen examenstof,
                               // kan meer mee dan enhanced for loop
    while(i.hasNext())    {
      System.out.println(i.next()); // next() heeft hier
                                    // reference type Object
    }
  }
}
```

Nieuwe notatie - generics

```
import java.util.*;
class CollectionApp {
   public static void main(String args[])  {
    Collection<String> c = new ArrayList<String>();
    c.add("Collection");
    c.add("array");
    c.add("Set");
    c.add("array");
    for(String naam : c)  // enhanced for loop
      System.out.println(naam);
    }
  }
}
```

De compiler controleert of toegevoegde elementen van het opgegeven datatype zijn. Het generic type komt zelf niet in de .class bestanden te staan (type erasure).

Verwijderen van een element

```
    // Collection<String> c = new ArrayList<String>(); ...
    while(c.contains("array")){
      c.remove("array");
    }

    // alternatief:
    Iterator<String> iterator = c.iterator();
    while(iterator.hasNext()){
      String s = iterator.next();
      if("array".equals(s)){
        iterator.remove();
      }
    }
// of:
  List<String> toRemove = new ArrayList<>();
  toRemove.add("array");
  list.removeAll(toRemove);
```

Autoboxing en unboxing

- Collecties kunnen alleen referentie typen bevatten
- Java kan automatisch wrapper class typen omzetten naar het primitieve type en andersom (sinds Java 5)

```
public static void main (String args[]){
  List<Float> prijzen = new ArrayList<Float>();
   prijzen.add(1.60F );
   prijzen.add(1.50F );
   float totaal=0;
   for(float p: prijzen) {
      totaal+=p;
   }
   System.out.println(totaal);
 }
```

Collection methods

```
Collection<String> c = new ArrayList<>(); ...
System.out.println(c.size()); // 0
c.add("test"); // voegt "test" toe
System.out.println(c.contains("test")); // true
c.remove("test"); // verwijder eerste object "test"
System.out.println(c.isEmpty()); // true
c.addAll(Arrays.asList("een","twee", "drie"));
System.out.println(c.size());  // 3
System.out.println(c.containsAll(Arrays.asList("drie", "twee",
"een"))); // true
c.removeAll(Arrays.asList("een", "twee"));
System.out.println(c.size()); // 1
c.clear();
System.out.println(c.isEmpty()); // true
```

List methods - met index

```
List<String> list = new ArrayList<>(); ...
list.add("A");   // [A]
list.add(0,"B"); // [B, A]
System.out.println(list.get(1));        // A
System.out.println(list.indexOf("A")); // 1
list.add(1,"C"); // [B, C, A]
list.remove(2);  // verwijder element op positie
list.set(1,"D"); // wijzig element op positie
```

Autoboxing en unboxing - vervolg

Sinds Java 5 kunnen primitieven impliciet of expliciet gecast worden naar hun wrapper class en andersom.

expliciet:

```
Integer a = (Integer)10;
int b = (int) a;
```

of impliciet:

```
Integer a = 10;
int b = a;
Float c = 10F;
System.out.println(b==c); // unboxing: true
System.out.println(a==c); // geen unboxing,
                          // compiler error, incomparable types
System.out.println(a<=c); // unboxing: true
```

Overloading en verbreding, autoboxing, varargs

Gegeven is de volgende aanroep:

```
doeIets(5);
```

en de volgende methoden:

```
public void doeIets(long l){...}
public void doeIets(Integer i){...}
public void doeIets(int... x){...}
```

De compiler zal dan eerst kiezen voor verbreden:	doeIets(long l);
Anders kiest de compiler autoboxing:	doeIets(Integer i);
Als het niet anders kan wordt varargs gekozen:	doeIets(int... x);

Generics en collecties

- objecten moeten van de opgegeven klasse zijn
- dit mag niet:
  - `Collection<Object> c = new ArrayList<String>();`

- om flexibel te kunnen zijn met datatype van elementen:
  - wildcards
  - wildcards met extends
  - parameterized methoden

Sinds Java 7 mag dit:

```
List<Integer> lijst = new ArrayList<Integer>();
```

vervangen worden door dit (diamond syntax):

```
List<Integer> lijst = new ArrayList<>();
```

De compiler zorgt er dan voor dat ook aan de rechterkant het juiste type staat.

Wildcards voor datatypen (geen examenstof)

Deze en volgende dia's over generic types worden niet getoetst in het examen

- De wildcard <?> staat voor een willekeurig type.
- Alleen handig als er geen elementtype-specifieke code nodig is.

```
public static void main(String args[]) {
  List<String> artikelen = new ArrayList<>();
  List<Integer> priemgetallen = new ArrayList<>();
  artikelen.add("schrijfblok");
  artikelen.add("balpen");
  priemGetallen.add(2);
  priemGetallen.add(3);
  toonInhoud(artikelen());
  toonInhoud(priemgetallen());
}
public static void toonInhoud(Collection<?> c) {
  Iterator<?> i = c.iterator();
  while(i.hasNext()) {
     System.out.println(i.next());
  }
}
```

Datatypen beperken met extends

- Met <? extends datatype> zijn elementen van opgegeven datatype of van subtype toegestaan
- Ook bij interface datatype extends gebruiken (geen implements)
- Geschikt voor element-specifieke code
- Geschikt om met een iterator te doorlopen, niet voor enhanced for loop

Voorbeeld: laat van een collectie met prijzen en een collectie met priemgetallen het totaal zien.

Datatypen beperken met extends - voorbeeld

```
  public static void main(String args[]) {
    Collection<Float> prijzen = new ArrayList<>();
    prijzen.add(4.5f);  prijzen.add(2.3f);
    toonTotaal(prijzen);
    Collection<Integer> priem = new ArrayList<>();
    priem.add(5);       priem.add(7);
    toonTotaal(priem);
  }
  public static void toonTotaal(Collection<? extends Number> c) {
    float totaal=0; Iterator<? extends Number> i = c.iterator();
    while(i.hasNext()){
      totaal+=i.next().floatValue();
    }
      System.out.println(totaal);
  }
// nadeel: geen controle of beide ? typen hetzelfde zijn
```

Parameterized methoden

- met <T extends datatype> kan een type T worden gedeclareerd, dat elders in de code gebruikt kan worden
- waarborg dat element-datatype van methode-parameter en methode-code gelijk is
- ook geschikt om met een enhanced for loop te doorlopen

Parameterized methoden - voorbeeld

```
public static <T extends Number> void toonTotaal(Collection<T> c) {
    float totaal=0;
    Iterator<T> i = c.iterator();
    while(i.hasNext()){
      totaal+=i.next().floatValue();
    }
      System.out.println(totaal);
}

public static <T extends Number> void toonTotaal(Collection<T> c) {
    float totaal=0;
    for(T element: c){
     totaal+=element.floatValue();
    }
   System.out.println(totaal);
}
```

toevoegen en ophalen van elementen

```
import java.util.*;
class Test {
   public static void main (String args[]){
      List<Integer> a = new ArrayList<>();
      List<Number>  b = new ArrayList<>();
      Collections.addAll(a,1,2,3,4);
      vul(a,b);
      System.out.println(b);
   }

   public static <T> void vul(List<? extends T> bron,
                              List<? super T> doel){
      for(T element:bron){
        doel.add(element);
      }
   }
}
```

Generics buiten collecties

- Ook aan klassen kunnen we type parameters meegeven
- Die typen worden binnen de klasse gebruikt
- Voorbeeld: interface Comparable.
  - SortedSet en SortedMap zijn collecties waarvan de elementen worden opgeslagen in hun natuurlijke volgorde.
  - Elementen dienen daartoe de interface Comparable te implementeren.

```
public interface Comparable<T> {
  int compareTo(T object);
}

// voorbeeld implementerende klasse
public class Geld implements Comparable<Geld>....

// voorbeeld als implementerende parameter
public <T extends Comparable<T>> void sorteer(T[] objecten){...}
```

Comparable implementeren

```
class Rechthoek implements Comparable<Rechthoek> {
  int lengte, breedte;
  Rechthoek(int lengte, int breedte) {
    this.lengte = lengte;
    this.breedte = breedte;
  }
  public String toString(){
    return "("+lengte+","+breedte+")";
  }
  public int oppervlakte(){
    return lengte*breedte;
  }
  public int compareTo(Rechthoek r){
    return oppervlakte() > r.oppervlakte()?  1:
           oppervlakte() < r.oppervlakte()? -1: 0;
  }
}
```

Comparable implementatie gebruiken

```
import java.util.*;
public class RechthoekApp {
  public static void main(String args[]) {
    List<Rechthoek> list = new ArrayList<Rechthoek>();
    Rechthoek r1 = new Rechthoek(3,5);  // oppervlakte = 15
    Rechthoek r2 = new Rechthoek(4,5);  // oppervlakte = 20
    Rechthoek r3 = new Rechthoek(3,6);  // oppervlakte = 18
    Rechthoek r4 = new Rechthoek(2,6);  // oppervlakte = 12
    list.add(r1);  // voegt r1 toe aan de list
    list.add(r2);  // voegt r2 toe aan de list
    list.add(r3);  // voegt r3 toe aan de list
    list.add(r4);  // voegt r4 toe aan de list
    Collections.sort(list); // sorteert de List m.b.v. compareTo()
    System.out.println(list);
  }
}
// uitvoer: [(2,6), (3,5), (3,6), (4,5)]
```

Alternatieve sortering: Comparator<T>

- Om elementen in afwijkende volgorde te kunnen
- Een comparator object kan meegegeven worden aan een aantal sorteer-methoden, zoals Collections.sort() en Arrays.sort()
- methode compare(T eerste, T tweede) vergelijkt eerste en tweede parameter van gelijk type

Comparator voorbeeld

```
class MijnComparator implements Comparator<String>{
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
  }
}
public class ComparatorApp {
  public static void main(String args[]) {
    List<String> lijst = Arrays.asList(args);
    Comparator<String> mijnComparator = new MijnComparator();
    Collections.sort(lijst, mijnComparator);
    System.out.println(lijst);
  }
}
// uitvoeren: java ComparatorApp Een twee Drie vier Vijf zes Zeven
// resultaat: [Drie, Een, twee, vier, Vijf, zes, Zeven]
```

H10: opgave 1 en 2

enums

```
public enum Maten {S, M, L, XL, XXL}
```

- Maten is hiermee een subklasse van java.lang.Enum
- Deze klasse implementeert o.a. Comparable
- Methoden o.a.:
  - toString(): geeft standaard de naam van de constante, kan worden overschreven
  - name(): geeft de naam van de constante, final methode
  - ordinal(): geeft de positie van de constante in de enum (0 based)
- Daarnaast heeft elke enum impliciet ook:
  - values(): geeft alle constanten als een array terug
  - valueOf(String): static methode om de enum bij deze naam op te halen

enum voorbeeld 1

```
enum Maten {S, M, L, XL, XXL};
public static void main(String args[]) {
  for(Maten m:Maten.values()) {
    System.out.println(m + ": " + omschrijving(m));
  }
}
 private static String omschrijving(Maten waarde) {
  switch(waarde)  {
    case S: return "Small";
    case M: return "Middle";
    case L: return "Large";
    case XL: return "Extra Large";
    case XXL: return "Extra Extra Large";
    default: return "Onbekende maat";
  }
} // nadeel: niet erg object georiënteerd
  // beter:  omschrijving aan enum zelf overlaten
```

enum voorbeeld 2

```
enum Maten {
  S("Small"), M("Middle"), L("Large"),
  XL("Extra Large"), XXL("Extra Extra Large");
  private final String maat;
  Maten(String maat) {  // constructor
    this.maat=maat;
  }
  public String toString() {
      return name()+"("+maat+")";
  }
}

public static void main(String args[]) {
  for(Maten m:Maten.values()) {
    System.out.println(m);
  }
}
```

enum voorbeeld 2 - uitvoer

```
java test
S(Small)
M(Middle)
L(Large)
XL(Extra Large)
XXL(Extra Extra Large)
```

H10: opgave 3

H11 - Garbage collection en foutafhandeling

Doelen:

- Weten wanneer garbage collection wordt toegepast
- Exceptions kunnen aanmaken
- Exceptions kunnen afvangen
- Gebruik kunnen maken van exception chaining

OCA boek: p36 t/m 38 (garbage collection),
H6 geheel (exceptions)

Garbage collection

- Vanaf het aanmaken leven objecten in het geheugen
- Als er geen referentie meer bestaat naar een object, kan het object worden opgeruimd
- Of en wanneer wordt opgeruimd wordt door Java zelf geregeld: garbage collection
- We kunnen niet zelf regelen wanneer garbage collection plaats vindt
- We kunnen wel zorgen dat objecten op tijd in aanmerking komen voor garbage collection
  - als null of een ander object wordt toegekend aan een variabele is de referentie verbroken
  - anders bepaalt de scope hoe lang een referentie naar het object blijft bestaan
- Garbage collection gebeurt in twee fasen:
  - objecten zonder referentie worden opgespoord en gelabeld
  - gelabelde objecten worden opgeruimd en geheugen vrijgegeven

performance verbeteren - 1

```
public class MijnProgramma {
  static GroteKlasse voorbeeld = new GroteKlasse();
  public static void main(String args[]){
    while(true){
      // doe iets
    }
    if("test".equals(args[0])) {
       System.out.println(voorbeeld.getInfo());
    }
  }
}

// groot object blijft nu lang ongebruikt in het geheugen
```

performance verbeteren - 2

```
public class MijnProgramma {
  public static void main(String args[]){
    while(true){
      // doe iets;
    }
    if("test".equals(args[0])) {
       System.out.println(info());
    }
  }
  private static String info(){
    GroteKlasse voorbeeld = new GroteKlasse();
    return voorbeeld.getInfo();
  }
}
// Nu wordt het object in de private methode info() aangemaakt.
// Daarna wordt het object kandidaat voor garbage collection.
```

Wanneer bestaat geen reference meer?

```
1. class Garbage {
2.     static Object object;
3.     public static void main(String args[]){
4.       Object a = new Object();  // wanneer kan dit object worden opgeruimd?
5.       Object b = new Object();  // wanneer kan dit object worden opgeruimd?
6.       if(check(a)==true){
7.         a = b;
8.       }
9.       else {
10.        b = a;
11.      }
12.      a = null;
13.      b = null;
14.    }
15.    public static boolean check(Object o){
16.      Garbage.object = o;
17.      return true;
18.    }
19.}
```

H11: opgave 1

Foutafhandeling

- Foutsituaties worden in Java exceptions genoemd
- Een exception wordt gerepresenteerd als een object van Throwable, of van een subklasse daarvan (zoals Exception, RuntimeException of Error)
- Een exception kan worden "opgegooid" (throw)
  of "afgevangen" (catch)
- Als een exception niet wordt afgevangen stopt het programma

Exception opgooien

```
class UpperApp {
  public static void main(String args[]) throws Exception {
    // foutsituatie als het aantal argumenten ongelijk is aan 1
    if(args.length!=1) {
      Exception e = new Exception("Gebruik: UpperApp <woord>");
      throw e;
    }
    System.out.println(args[0].toUpperCase());
  }
}

// uitvoeren:
java UpperApp
Exception in thread "main" java.lang.Exception: Gebruik: UpperApp
<woord>
        at UpperApp.main(UpperApp.java:7)
```

try en catch - voorbeeld

```
try {
  // code waarbinnen mogelijk een FileNotFoundException
  // of een IOException kan worden opgegooid
}
catch (FileNotFoundException fe)  {
  // code om fe af te handelen
}
catch(IOException ie) {
  // code om ie af te handelen
}
finally {
  // code die altijd moet worden uitgevoerd
  // bijvoorbeeld om een openstaand bestand te sluiten
}
```

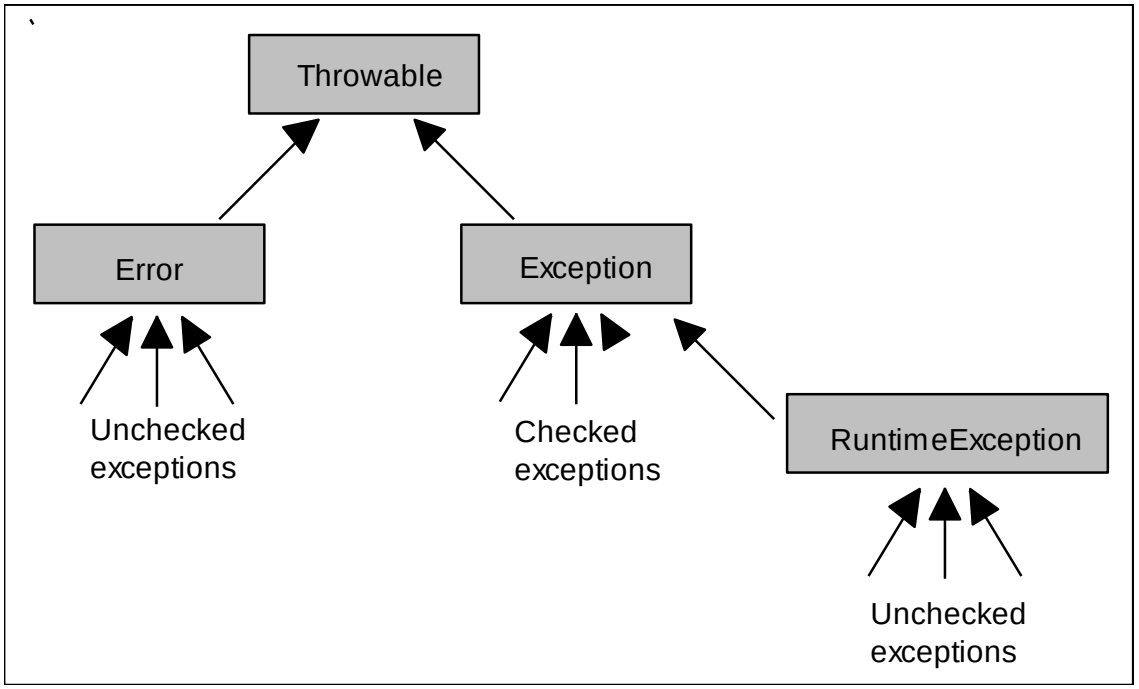
Exceptions afvangen: try en catch

- een **exception handler** is een speciaal stuk code, bedoeld om fouten af te vangen
- code waar een fout in kan optreden komen in een **try** blok
- exception handler bestaat tenminste uit een try blok en een **catch** blok, of een try blok en een **finally** blok
- onder het try blok kunnen meerdere catch blokken voorkomen
- elk catch blok kan één (of meer - sinds java 7) soorten fouten afvangen
- er kan hooguit één **finally** blok worden toegevoegd, onderaan de exception: dit blok wordt altijd uitgevoerd, of er nu wel of geen fout optreedt, tenzij de code voortijdig wordt afgebroken met System.exit(int status)
- tussen de try, catch en finally blokken mag geen code voorkomen
- als code uit het try blok geen fout oplevert wordt onder de catch blokken doorgegaan met een eventueel finally blok, daarna met de rest van de code
- als code wel een fout oplevert wordt gezocht naar het eerste catch blok die de fout kan opvangen

pas op volgorde van catch blokken

```
try {
  // code
}
catch (Exception e){
  // code om e af te handelen
}
catch (FileNotFoundException fe){
  // code om fe af te handelen
}
catch(IOException ie){
  // code om ie af te handelen
}
// fout bij compileren:
//
// exception java.io.FileNotFoundException has already been caught
// exception java.io.IOException has already been caught
// 2 errors
```

Soorten exceptions



- Throwable: hoofdklasse van alles wat met throw kan worden opgegooid
- Exception: checked exceptions - worden tijdens het compileren gechecked en moeten worden afgevangen met catch, of in declaratie van methode worden genoemd met throws <Exception>
- RuntimeException: unchecked exception - worden runtime gechecked en hoeven niet te worden afgevangen, dienen in de code te worden voorkomen (bijv. NullPointerException)
- Error: unchecked exception - fatale fouten die niet afgevangen worden omdat er in deze gevallen niks meer gered kan worden.

RuntimeExceptions

Naam	*Treedt wanneer op?*
ArrayIndexOutOfBounds-Exception	Als verwezen wordt naar een index van een Array buiten de lengte van de array.
ArithmeticException	bij een deling door 0.
ClassCastException	Als geprobeerd wordt een object naar een subklasse te casten waar het geen instantie van is.
IllegalArgumentException	Als een parameterwaarde van een methode niet geldig is.
IllegalStateException	Als een methode op een verkeerd moment wordt aangeroepen, omdat de instantie (nog) niet de juiste state heeft. Zoals bijvoorbeeld de methode gasGeven() van een Auto instantie niet kan worden uitgevoerd voordat starten() is aangeroepen.
NullPointerException	Als een methode wordt aangeroepen vanuit een variabele die niet (meer) naar een object verwijst.
NumberFormatException	Als een ongeldige String wordt gebruikt om te converteren naar een nummer.

Exceptions

Naam	*Treedt wanneer op?*
ClassNotFoundException	Sommige methoden kunnen een klasse object ophalen bij een als String ingevoerde naam, zoals de Class.forName() methode. De ClassNotFoundException treedt op als de gevraagde klasse niet kan worden gevonden.
IOException	Bij het lezen en schrijven naar bestanden, bijvoorbeeld als het betreffende bestand niet kan worden gevonden.
NoSuchMethodException	Bij een aanroep van een niet bestaande methode.
SQLException	Bij een database fout.

Errors

Naam	*Treedt wanneer op?*
ExceptionInInitializerError	Als zich een fout heeft voorgedaan bij het uitvoeren van een static initializer block
StackOverflowError	Als code die recursief wordt aangeroepen te diep genest wordt.
NoClassDefFoundError	Als het .class bestand van een gebruikte klasse runtime niet gevonden kan worden. Deze fout treedt ook op als een spelfout wordt gemaakt bij het aanroepen van een applicatie.

Eigen exceptions

```
class RangeException extends Exception {
  RangeException(String s) {
    super(s);
  }
}

class PrecisieException extends Exception {
  PrecisieException(String s) {
    super(s);
  }
}
```

public Exception(String message)

Constructs a new exception with the specified detail message.

Parameters:

message - the detail message. The detail message is saved for later retrieval by the Throwable.getMessage() method.

Veel gebruikte methoden van Throwable

Methode	*Omschrijving*
`Throwable getCause()`	Geeft de fout die deze fout heeft veroorzaakt, of null als er geen veroorzakende fout is.
`String getMessage()`	Geeft de foutboodschap die bij het opwerpen van de fout is meegegeven.
`void printStackTrace()`	Geeft gedetailleerde informatie over de plek waar een fout is opgeworpen, en waar de betreffende methode is aangeroepen.
`String toString()`	Geeft de fout in de form van <klassenaam>: <foutboodschap>

Opgooien van eigen exceptionss

```
static String getBedrag(double bedrag) throws RangeException,
PrecisieException  {
    if(bedrag<=0 || bedrag>=100000)
      throw new RangeException("Bedrag niet tussen 0 en 100000.");
    BigDecimal decimal = new BigDecimal(Double.toString(bedrag));
    decimal=decimal.movePointRight(2);
    if(decimal.scale() > 0 )
      throw new PrecisieException(
                "Bedrag heeft meer dan twee decimalen.");
    Locale local = new Locale("nl","NL", "EURO");
    NumberFormat form = NumberFormat.getCurrencyInstance(local);
    return form.format(bedrag);
  }
```

Opvangen van eigen exceptions

```
public static void main(String args[]){ // geen throws: zelf opvangen
    double bedrag=0;
    try {
      bedrag = Double.parseDouble(args[0]);
      System.out.println("Bedrag in Euro's: " + getBedrag(bedrag));
    }
    catch(RangeException e){ System.err.println(e.getMessage()); }
    catch(PrecisieException e){  e.printStackTrace(); }
    catch(Exception e){ e.printStackTrace(); }
    finally {
      System.out.println("Bedrag was: " + bedrag);
    }
  }
```

printStackTrace()

Deze methode retourneert de **stack trace**: een gedetailleerd overzicht van de methoden in de code die betrokken zijn bij het opgooien van de fout.

```
void a(){
  b();
}
void b(){
  c();
}
void c(){
  // code met fout.
}
```

Stack trace geeft aan:

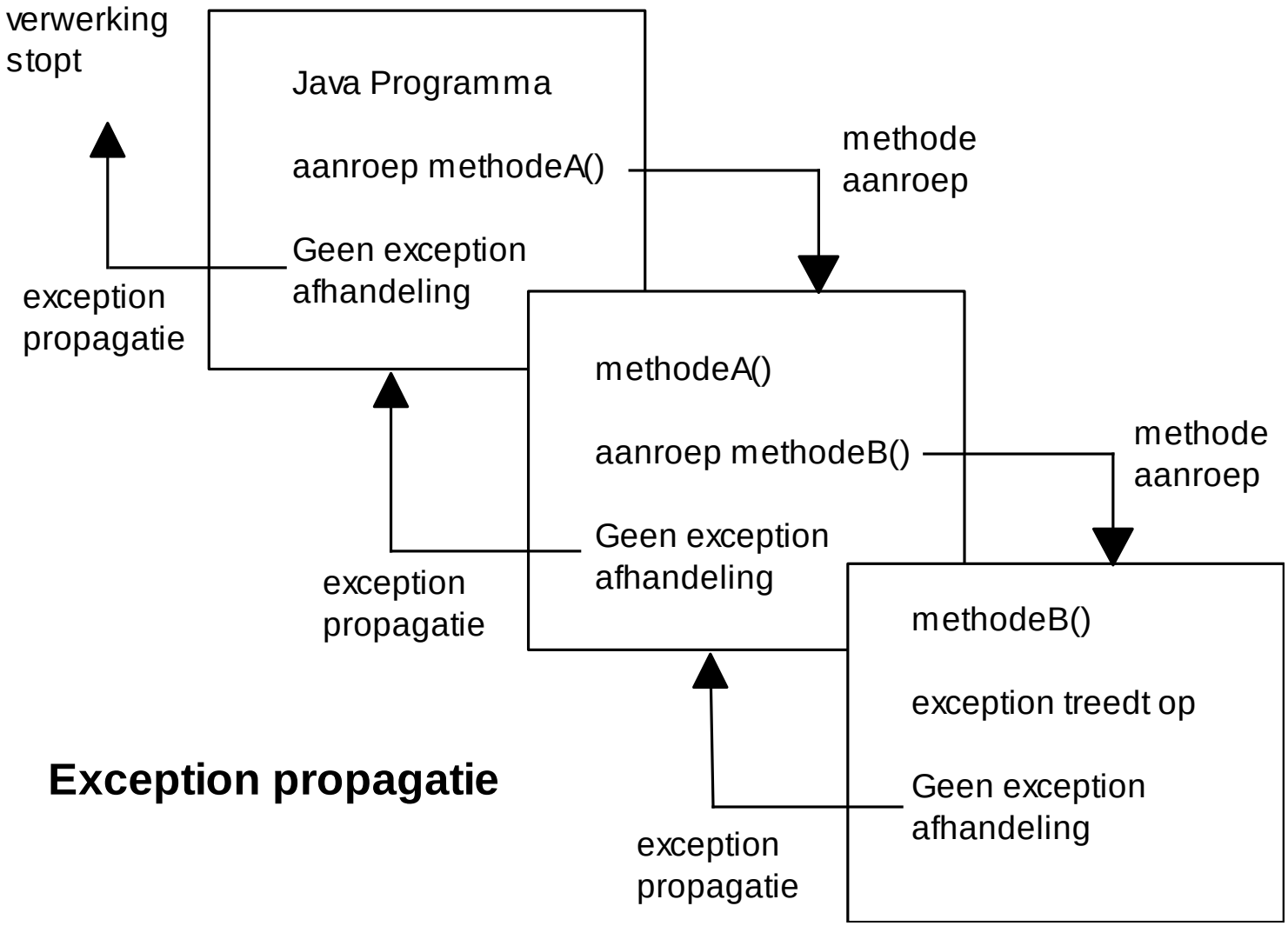
- welke fout zich waar in c() voordeed
- waar dat veroorzaakt is in b()
- waar dat veroorzaakt is in a()

voorbeeld uitvoer printStackTrace()

```
java TryCatchApp 1234.567
PrecisieException: Bedrag heeft meer dan twee decimalen.
        at TryCatchApp.getBedrag(TryCatchApp.java:45)
        at TryCatchApp.main(TryCatchApp.java:17)
Bedrag was: 1234.567
```

- Eerst wordt aangegeven waar de fout is opgetreden (regel 45, in methode getBedrag()).
- Daarna wordt aangegeven waar getBedrag() is aangeroepen (regel 17, in de main() methode)

propagatie



Exception propagatie

Excepties opnieuw opgooien

```
public static void main(String args[]) {
  int deler = 0, uitkomst = 0;
  try{
     deler = Integer.parseInt(args[0]);
     uitkomst = 100 / deler;
  }
  catch(ArrayIndexOutOfBoundsException e){
     uitkomst = 100;
  }
  catch(ArithmeticException e) {
   uitkomst = Integer.MAX_VALUE;
   throw e;
  }
  finally{
    System.out.println(uitkomst);
  }
}
```

Exception chaining

- als reactie op een fout kan een andere exception worden opgeworpen
- standaard zien we dan in de stack trace alleen waar de laatste fout heeft plaats gevonden

```
...
   catch(ArithmeticException e)
   {
    uitkomst = Integer.MAX_VALUE;
    throw new DelerIsNulException("De deler mag niet nul zijn.");
   }
   ...

// Informatie over de oorsprong (de ArithmeticException)
// zijn we nu kwijt
```

Exception chaining - vervolg

```
class DelerIsNulException extends RuntimeException
{
  public DelerIsNulException(String s, Throwable t) {
    super(s,t);
  }
}
...
throw new DelerIsNulException("De deler mag niet nul zijn.", e);
...
java DeelApp 0
2147483647
Exception in thread "main" DelerIsNulException: Delen door nul niet
toegestaan
.
        at DeelApp.main(DeelApp.java:19)
Caused by:java.lang.ArithmeticException: / by zero
        at DeelApp.main(DeelApp.java:10)
```

Exceptions en overriding

Gegeven:

```
class A {
  void m() throws IOException{...}
}

class B extends A {
  @Override
  void m() ....
}
```

In B mag m() nu hooguit IOException, een subklasse daarvan, en/of unchecked exceptions opgooien.

Of m() gooit in B niets op, dat mag ook.

Overweging: A a = new B(); a.m();

Exceptions en constructors

Gegeven:

```
class A {
  A() throws IOException{...}
}

class B extends A {
   B() ....
}
```

De constructor van B() moet nu in ieder geval IOException opgooien of een superklasse daarvan. Daaraan kunnen andere exceptions worden toegevoegd.

Overweging: In B() wordt impliciet super() aangeroepen, als eerste regel. Er kan dus geen try-catch block omheen worden gezet.

Exceptions en static initializers

Een static block mag geen checked exceptions opgooien.

Unchecked exceptions mogen wel worden opgegooid,
maar kunnen niet worden afgevangen.

Een runtime exception in een static block veroorzaakt een ExceptionInInitializerError

H11: opgave 2

h12: Lambda expressies

- functionele interfaces
- doorgeven van functionaliteit
- kortere code
- nieuwe syntax

OCA boek: 208 t/m 215

Comparator implementeren - van Java 7 naar 8

Wat in onderstaande code is de logica die moet worden toegepast?
Wat is noodzakelijke extra code (boilerplate code)?

```
class MijnComparator implements Comparator<String>{
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
  }
}

public class ComparatorApp {
  public static void main(String args[]) {
    List<String> lijst = Arrays.asList(args); // unmodifiable List
    Comparator<String> mijnComparator = new MijnComparator();
    Collections.sort(lijst, mijnComparator);
    System.out.println(lijst);
  }
}
```

Korter in Java 7: anonieme inner class

```
public class ComparatorApp {
  public static void main(String args[]) {
    List<String> lijst = Arrays.asList(args);
    Collections.sort(lijst, new Comparator<String>(){
          public int compare(String s1, String s2) {
            return s1.compareToIgnoreCase(s2);
          }
    });
    System.out.println(lijst);
  }
}
```

Kan dat nog korter?

Wat in deze code zou de compiler ook kunnen bedenken?
Wat kun je niet aan de compiler overlaten?

```
Collections.sort(lijst, new Comparator<String>(){
      public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
      }
});
```

- Collections.sort van een List<String>, dus tweede argument moet een Comparator<String> zijn
- De enige methode in Comparator<String> is compare()
- Returntype en datatypen van de argumenten zijn ook bekend
- Wat in die methode moet gebeuren, dat moeten we zelf bepalen

Verder verkorten in Java 8

```
Collections.sort(lijst, new Comparator()<String>{
          public int compare(String s1, String s2) {
            return s1.compareToIgnoreCase(s2);
          }
    });

// Comparator heeft één methode, naam kan compiler afleiden
Collections.sort(lijst,
   (String s1, String s2) -> {return s1.compareToIgnoreCase(s2);});

// type inference: datatypen van s1 en s2 kan compiler afleiden
Collections.sort(lijst,
(s1, s2) -> {return s1.compareToIgnoreCase(s2);});

// enige statement moet het return statement zijn, "return" kan weg
Collections.sort(lijst,(s1, s2) -> s1.compareToIgnoreCase(s2));
```

Voordelen van lambda expressies

- minder code: minder kans op fouten
- makkelijker te lezen
- beter te onderhouden
- bevordert functionele benadering van programmeren
  - beperkt afhankelijkheid van variabelen en state
  - uitkomst van functies is daardoor beter te begrijpen en voorspellen

Syntax

- lambda expressie specificeert de enige abstracte methode van een interface
- eerst parameters tussen haakjes
  - datatype parameters kan vaak achterwege blijven
  - zonder parameters: alleen haakjes
  - één parameter: haakjes optioneel
- dan pijltje ->
- dan:
  - expressie dat overeenkomt met return type, zonder "return" en ;
  - of blok met één of meer statements, elk statement met ;

Voorbeeld - Runnable

Een thread is een deel programmacode dat parallel draait. Zo'n thread moet Runnable implementeren, met de void methode run();

oud:

```
Runnable thread = new Runnable(){
  public void run(){
     System.out.println("Op oude manier");
  }
};
```

nieuw:

```
Runnable thread = ()-> System.out.println("op nieuwe manier");
```

Voorbeeld - ActionListener

GUI applicaties in swing maken gebruik van listeners om events af te vangen. De interface ActionListener heeft één methode:

```
public void actionPerformed(ActionEvent event){...}
```

Voorbeeld oude manier:

```
knop.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event){
      System.out.println("Er is op een knop gedrukt");
   }
});
```

Voorbeeld met gebruik van lambda expressie:

```
knop.addActionListener(
          event -> System.out.println("Er is op een knop gedrukt"));
```

Gebruik van waarden

Oud: alleen **final** waarden uit omliggende methode kunnen in inner class worden gebruikt:

```
final String naam = getNaam();
knop.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event){
      System.out.println("Hallo " + naam);
   }
});
```

Nieuw: alleen lokale variabelen die ***effectief* final** zijn kunnen in inner classes, en dus ook in lambda expressie worden gebruikt.

```
String naam = getNaam();
knop.addActionListener(event -> System.out.println("Hallo " + naam));
```

Functionele interfaces (examen: alleen Predicate)

- Lambda expressies kunnen gebruikt worden op de plek van (implementaties van) "functionele" interfaces:
  - deze hebben één abstracte methode
  - daarnaast eventueel default en/of static methoden
  - maken geen gebruik van internal state

- Annotatie @FunctionalInterface:
  - geeft aan dat de interface functioneel is bedoeld
  - compiler checkt of er één abstracte methode is

- Veel nieuwe functionele interfaces in java.util.function, o.a.:
  - **Predicate<T>: boolean test(T t)** - test geldigheid van invoer
  - **Consumer<T>: void accept(T t)** - voert iets uit, geeft niks terug
  - **Function<T,R>: R apply(T t)** - bewerkt invoer en geeft iets terug
  - **Supplier<T>: T get()** - genereert een resultaat zonder invoer
  - **BinaryOperator<T>: T apply(T t1, T t2)** - bewerkt twee invoeren tot een resultaat

Predicate<T> - boolean test(T t)

```
import java.util.function.*;
import java.util.*;
class Test {
   public static void main (String args[]){
     System.out.println(filter(args, s -> s.length()>3));
     System.out.println(filter(args, s -> s.indexOf("i")>=0));
   }
   private static List<String> filter(String[] woorden, Predicate<String> p){
      List<String> lijst = new ArrayList<>();
      for(String s: woorden){
        if(p.test(s)){
          lijst.add(s);
        }
      }
      return lijst;
   }
}
aanroep: java Test een twee drie vier
uitvoer: [twee, drie, vier]
         [drie, vier]
```

Consumer<T> - void accept(T t)

```
// methode om geselecteerde artikelen te bewerken
public static void bewerk(List<Artikel> lijst,
                          Predicate<Artikel> p, // selectie
                          Consumer<Artikel> c){ // bewerking
    for(Artikel a:lijst){
       if(p.test(a))
          c.accept(a);
    }
 }

public static void main(String args[]){
 [...]
 bewerk(artikelen,
        a -> a.laatstVerkocht()>30,
        a -> a.setPrijs(a.getPrijs() * .9));
}
```

Function<T,R> - R apply(T t)

```
import java.util.function.*;
class FunctionApp {
   public static void main(String args[]){
        String woord = args[0];
        System.out.println(bewerk(woord,x->x.length()));
        System.out.println(bewerk(woord,x->x.indexOf("i")));
   }

   static int bewerk (String invoer,
                      Function<String, Integer> functie ){
     int uitvoer = functie.apply(invoer);
     return uitvoer;
   }
}
```

Supplier<T> - T get()

```
class A {
  private int id;
  private static int hoogste=0;

  private A(int id){
   this.id=id;
  }

  public String toString(){
    return String.format("A(%s)",id);
  }

  static Supplier<A> getNextA = () -> new A(++hoogste);

  static Function<Integer,A> getA = x -> new A(x>hoogste?hoogste=x:++hoogste);
}
```

Supplier<T> - vervolg

```
class Test {
   public static void main(String args[]){
       for (int i=0;i<5;i++){
         System.out.println(A.getNextA.get()); // genereer id: Supplier
      }
      System.out.println(A.getA.apply(10));   // geef id mee: Function
      System.out.println(A.getNextA.get());
   }
}

uitvoer:
A(1)
A(2)
A(3)
A(4)
A(5)
A(10)
A(11)
```

BinaryOperator<T> T apply(T t1, T t2)

```
import java.util.function.BinaryOperator;
class BinaryOperatorApp {

  public static void main(String args[]){
    System.out.println(join(1,100,(a,b)->a+b));
    System.out.println(join(1,7,(a,b)->a*b));
  }

  static long join(long start, long end, BinaryOperator<Long> bo){
    long result = start;
    for(long val=start+1;val<=end;val++){
      result = bo.apply(result, val);
    }
    return result;
  }
}
// uivoer:
5050
5040
```

default methoden - Predicate

Veel functionele interfaces kunnen gecombineerd worden met default methoden.

Predicate<T> is bedoeld om voorwaarden te testen.

Deze interface bevat de methoden and(), or() en negate() om voorwaarden te combineren.

```
Predicate<Integer> p = x -> x > 10;
p = p.and(x -> x%2==0);        // logical and
Predicate<Integer> p2 = x -> x <=10;
p2 = p2.and(x -> x%2==1);
p=p.or(p2);                    // logical or
for(int x=5;x<15;x++){
  if(p.negate().test(x)){      // logical not
     System.out.println(x);
  }
}
```

default methoden - Function

Het resultaat van de ene functie kan gebruikt worden als invoer van de andere functie:

- f1.andThen(f2): voer eerst f1 uit, geef resultaat door aan f2
- f1.compose(f2): voer eerst f2 uit, geef resultaat door aan f1

```
Function<String,Integer>  toInt  = x -> Integer.parseInt(x);
Function<Integer,Integer> dubbel = x -> x*2;
toInt = toInt.andThen(dubbel);
toInt = dubbel.compose(toInt);
System.out.println(toInt.apply(args[0]));

// aanroep: java Test 5
// uitvoer: 20
```

default methoden - Consumer

Meerdere Consumers kunnen aaneengeschakeld worden met andThen():

```
Consumer<Integer> toon = x -> System.out.println(x);
Consumer<Integer> dubbel = x -> System.out.println(x*2);

dubbel = toon.andThen(dubbel);
dubbel.accept(5);

// aanroep: java Test 5
// uitvoer: 5
//          10
```

default methoden - BinaryOperator

Het resultaat BinaryOperator kan met andThen door een Function worden bewerkt.

```
BinaryOperator<Integer> plus = (x,y) -> x + y;
Function <Integer,Integer> dubbel = x -> x * 2;
System.out.println(plus.andThen(dubbel).apply(3,4));

// resultaat: 14
```

method references

- Binnen een lambda expressie wordt vaak gebruik gemaakt van methoden of constructors.
- Deze aanroepen kunnen vaak ook vervangen worden door method references
- Zo'n reference kan gebruikt worden, als de methode dezelfde signatuur en return type heeft als de functionele interface
- method reference bestaat uit:
  - klassenaam::methodenaam - static methoden,
    of methoden die parameter(s) van lambda gebruiken
  - instantie::methodenaam - methode te gebruiken vanuit variabele

method references - voorbeeld

Deze methoden voeren iets uit en tonen het resultaat.

```
static <T,R> void bewerk(T t , Function<T,R> functie){
	System.out.println(functie.apply(t));
}
static <T> void maak(Supplier<T> supplier){
	System.out.println(supplier.get());
}
static <T,R> void array(T t , Function<T,R[]> functie){
	System.out.println(Arrays.toString(functie.apply(t)));
}
static <T> void voerUit(T t, Consumer<T> consumer){
	consumer.accept(t);
}
static <T> void pasToe(T t1, T t2, BinaryOperator<T> operator) {
	System.out.println(operator.apply(t1,t2));
}
```

method references - vervolg

```
bewerk("abc", x -> x.toUpperCase());
bewerk("abc", String::toUpperCase);          // ABC - x is object
String tekst = "test";
bewerk("je", x -> tekst.concat(x));
bewerk("je", tekst::concat);                 // testje - tekst is object
                                             //          x is argument
bewerk(10.5, x -> Math.sqrt(x));
bewerk(10.5, Math::sqrt);                    // 3.24037034920393
maak(()->new Object());
maak(Object::new);                           // java.lang.Object@23fc625e
array(2, x -> new String[x]);
array(2, String[]::new);                     // [null, null]
voerUit(tekst,x -> System.out.println(x));
voerUit(tekst,System.out::println);          // test - x is argument
pasToe(3,4,(x,y) -> Math.max(x,y));
pasToe(3,4,Math::max);                       // 4 - x en y als argumenten
pasToe("test","je",(x,y) -> x.concat(y));
pasToe("test","je",String::concat);          // testje - x is object,
                                             //          y is argument
```

H12: opgave 1 en 2

H13: date/time API

- Nieuw in Java 8
- Vereenvoudigde manier om met datum, tijd en periodes om te gaan
- java.time (en subpackages)

OCA boek: p138 t/m 151

Oud: Date, Calendar, SimpleDateFormat

- Voorheen: new Date() = huidig moment, Calendar om met datums te rekenen, SimpleDateFormat om datums / strings naar elkaar te converteren.
- Date en SimpleDateFormat zijn niet thread safe
- API is niet intuïtief

Nieuw: java.time

- Immutable classes – thread safe
- Naar keuze met/zonder tijd, met/zonder tijdzone-informatie
- Consistente naamgeving van methoden voor aanmaken, aanpassen en formatteren van date / time objecten
- Goed leesbaar

Method naming conventions

Prefix	Method Type	Use
of	static factory	Creates an instance where the factory is primarily validating the input parameters, not converting them.
from	static factory	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
parse	static factory	Parses the input string to produce an instance of the target class.
format	instance	Uses the specified formatter to format the values in the temporal object to produce a string.
get	instance	Returns a part of the state of the target object.
is	instance	Queries the state of the target object.
with	instance	Returns a copy of the target object with one element changed; this is the immutable equivalent to a set method on a JavaBean.
plus	instance	Returns a copy of the target object with an amount of time added.
minus	instance	Returns a copy of the target object with an amount of time subtracted.
to	instance	Converts this object to another type.
at	instance	Combines this object with another.

And now ...

```
import java.time.*;

public class Now {
  public static void main(String args[]){
    LocalDate vandaag = LocalDate.now();
    LocalTime nu = LocalTime.now();
    LocalDateTime nuVandaag = LocalDateTime.now();
    OffsetDateTime nuInZone = OffsetDateTime.now();
    ZonedDateTime nuInZoneNaam = ZonedDateTime.now();
    System.out.println(vandaag);          // 2015-10-26
    System.out.println(nu);               // 14:26:08.161
    System.out.println(nuVandaag);        // 2015-10-26T14:26:08.161
    System.out.println(nuInZone);         // 2015-10-26T14:26:08.161+01:00
    System.out.println(nuInZoneNaam);
                          // 2015-10-26T14:26:08.161+01:00[Europe/Berlin]
  }
}
```

Datum/tijd opgeven: of(), parse()

```
LocalDate datum = LocalDate.of(2015,9,25) // 1-based
// let op: gebruik geen 09, wordt als octaal getal gelezen
LocalDate dagLater = LocalDate.of(2015,Month.SEPTEMBER, 26);
// enum java.time.Month
LocalTime lunch = LocalTime.of(12,0);
// uren (0-23), minuten [, seconden [, nanoseconden]]


LocalDate datumUitTekst = LocalDate.parse("2015-09-27");
LocalTime theeTijd = LocalTime.parse("14:30:00");
```

Type wijzigen: from() en at()

```
LocalDate vandaag = LocalDate.now();
YearMonth ym = YearMonth.from(vandaag); // alle info beschikbaar
LocalDateTime ldt = vandaag.atTime(LocalTime.now()); //info toevoegen
ZonedDateTime zdt = vandaag.atStartOfDay(ZoneId.systemDefault());

System.out.println(vandaag);
System.out.println(ym);
System.out.println(ldt);
System.out.println(zdt);
```

```
2015-11-12
2015-11
2015-11-12T12:10:31.448
2015-11-12T00:00+01:00[Europe/Berlin]
```

Datum/tijd manipuleren: plus() minus()

```
import java.time.*;

class PlusMinus {
  public static void main(String args[]){
    LocalDate datum = LocalDate.of(2016,2,28); // schrikkeljaar
    datum = datum.plusDays(1);
    System.out.println(datum);
    LocalTime tijd = LocalTime.of(22,30,0);    // 22:30 uur
    tijd = tijd.plusHours(3)
               .minusMinutes(5)
               .plusSeconds(10);               // chained aanroep
    System.out.println(tijd);
  }
}
```

Datum/tijd formatteren: DateTimeFormatter

```
import java.time.*;
import java.time.format.*;

....
LocalDateTime nu = LocalDateTime.now();
DateTimeFormatter dtf1 = DateTimeFormatter
  .ofPattern("dd MMMM yyyy HH:mm");
DateTimeFormatter dtf2 = DateTimeFormatter
  .ofLocalizedDate(FormatStyle.SHORT); // MEDIUM / LONG / FULL
System.out.println(nu.format(dtf1));
System.out.println(nu.format(dtf2));
```

```
Het is nu: 12 november 2015 12:41
Het is nu: 12-11-15
Het is nu: 12-nov-2015
Het is nu: 12 november 2015
Het is nu: donderdag 12 november 2015
```

Aanpassen: Adjusters

```
LocalDateTime nu = LocalDateTime.now();
    DateTimeFormatter dtf = DateTimeFormatter
                            .ofLocalizedDate(FormatStyle.LONG);
    DateTimeFormatter dtft = DateTimeFormatter
.ofLocalizedDateTime(FormatStyle.LONG,FormatStyle.SHORT);
LocalDateTime volgendeWoensdag =
       nu.with(TemporalAdjusters.next(DayOfWeek.WEDNESDAY))
         .minusHours(2);
LocalDate eersteMaandagVolgendeMaand = LocalDate.from(nu)
          .with(TemporalAdjusters.firstDayOfNextMonth()
          .with(TemporalAdjusters
               .firstInMonth(DayOfWeek.MONDAY));
System.out.println("Woensdag, 2 uur eerder: " + dtft.format(volgendeWoensdag));
System.out.println("Eerste maandag volgende maand: " +
dtf.format(eersteMaandagVolgendeMaand));
```

```
Woensdag, 2 uur eerder: 18 november 2015 11:17
Eerste maandag volgende maand: 7 december 2015
```

Periodes

```
class Periods {
   public static void main(String args[]){
     Period tweeWekelijks = Period.ofWeeks(2);
     Period vierWekelijks = Period.ofWeeks(4);
     kalender(LocalDate.of(2016,1,12), tweeWekelijks, "restafval");
     kalender(LocalDate.of(2016,1,5),  tweeWekelijks, "GFT");
     kalender(LocalDate.of(2016,1,28), vierWekelijks, "papier");
   }
   public static void kalender (LocalDate start,
                                Period periode,
                                String rolemmer){
       System.out.println(rolemmer);
       LocalDate dag = start;
       while(dag.isBefore(LocalDate.of(2017,1,1))){
        System.out.println(dag);
        dag = dag.plus(periode);
       }
   }
}
```

H13: opgaven

Einde van de cursus

Gelieve de evaluatie in te vullen op mijn.vijfhart.nl

Via de online leeromgeving kunnen nog vragen worden gesteld.

Veel succes en graag tot ziens bij Vijfhart!