Introductie tot Java voor MHP
Professionele Bachelor MCT

koen.soontjens@kdg.be

16 januari 2010

Inhoudsopgave

Hoofdstuk 1

Introductie tot Java

1.1 Inleiding

Java lijkt wat betreft de syntax heel erg op C. Java is echter niet "compatibel" met C, er zijn ook belangrijke verschillen. Toch zal je merken dat heel wat principes van C behouden blijven.
Een belangrijk verschil met het programmeren in talen zoals VB.NET is dat ALLES in Java hoofdletter gevoelig is. Dit houdt in dat de variabele "varc" een andere variabele is dan de variabele "Varc" en nog een andere variabele is dan "VARC", terwijl in Visual Basic het om dezelfde variabele zou gaan is het in Java (net als in C) drie keer een andere variabele. Een tweede verschil met het programmeren in VB.NET is dat elk statement (commando) in Java afgesloten wordt met een ";". In VB.NET geven de regeleindes de scheiding tussen commando's aan. In Java niet, in Java mogen in principe alle commando's op één regel geplaatst worden, zolang ze gescheiden worden door puntkomma's.

1.2 Een voorbeeld programma

Een eenvoudig Java programma lijkt heel fel op een C programma. Hieronder vind je een C programma en een Java programma dat hetzelfde doet.

```c
//bestandsnaam: EersteProg.c
#include <stdio.h>

void main()
{
    printf("Dit is mijn eerste C programma\n");
}
```

```java
//bestandsnaam: EersteProg.java
import java.lang.*;

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

```
    {
        System.out.println("Dit is mijn eerste Java programma\n");
    }
}
```

Wanneer we beide programma's vergelijken zijn er duidelijk gelijkenissen:
#include <stdio.h> wordt in het Java programma **import java.lang.*;**
Merk op dat de # wegvalt en de regel met een ; wordt afgesloten in Java. De bedoeling van beide regels is dat je later in het programma een functie kunt gebruiken om tekst af te drukken (printf in C, System.out.println in Java).
Hoewel de #include en import commando's ongeveer hetzelfde doel hebben, is hun interne werking verschillend:
In C zal het #include commando een bestand zoeken (stdio.h) en dit bovenaan jouw broncode "plakken." (Het bestand "stdio.h" bevat ondermeer de definitie van de printf-functie).
In Java geef je met "import java.lang.*" aan dat je alle objecten die zich in de mappenstructuur onder java.lang bevinden (zoals "System") verkort wil noteren. Je zou ook de import kunnen weglaten en telkens in je code noteren: java.lang.System.out.println("tekst");
Maar om de "verkorte" schrijfwijze te kunnen gebruiken (System.out.println) zet je bovenaan het programma: import java.lang.*;
Overigens gaat het hier om een uitzondering: de import java.lang.*; hoef je eigenlijk niet te noteren, Java importeert de klassen onder java.lang automatisch, maar dit geldt niet voor de overige klassen uit de Java bibliotheek.

Verder valt op dat we in beide gevallen een main-functie hebben die uitgevoerd wordt wanneer het programma gestart wordt. In Java zit deze functie echter in een klasse. De klasse die de main functie bevat moet dezelfde naam hebben als het bestand (hoofdletter gevoelig, in dit geval "EersteProg")
String args[] (tussen de haakjes bij main) is parameter die wordt meegegeven aan de main-functie (main-methode). Op deze manier wordt een array van Strings doorgegeven aan de main-functie. Bij het starten van het programma kan je hiermee een aantal parameters doorgeven aan de main functie.

Binnenin de main-functie heb je in beide gevallen een functie die een tekst afdrukt. In C is dit de printf-functie, in Java is dit de functie System.out.println. Hoewel beide functies in essentie tekst afdrukken zijn er toch verschillen: zo is het gebruik van formaat-aanduiders (%d, %s, %x, %c) enkel in C toegestaan en niet in Java. In Java gebruik je een +-teken om een aantal Strings aan elkaar te plakken. Bijvoorbeeld:

```
System.out.println ("Dit is een " + "test" + arg[0]);
```

1.3 Het compileren en runnen van een Java programma

Elke JDK (Java Development Kit) is voorzien van een Java compiler. Deze compiler kan je vanuit een commando-prompt oproepen:

1. Alleerst open je een command prompt:
   Windows-toets+R → cmd.exe
2. Ga naar de bin-map van de JDK:
   cd \Program Files\Java\ jdk1.6.0_07\bin
3. Start de compiler, geef op welk .java-bestand je wil compileren:
   javac EersteProg.java

   Hierbij wordt er vanuit gegaan de het bronbestand zich in de map \Program Files\Java\ jdk1.6.0_07\bin bevindt.

Een Java-compiler geeft geen .exe bestand (dat processorafhankelijker machinetaalinstructies bevat) zoals een C-compiler. In de plaats daarvan wordt er echter een **.class** bestand gemaakt dat de uitvoerbare code bevat. Deze code bestaat uit processoronafhankelijke "bytecodes" die door een Java-interpreter of een Java-Just-In-Time compiler uitgevoerd worden. Vanuit de commando prompt run je het programma zo:

java EersteProg

De **.java** files worden dus gecompileerd met behulp van het **javac**-commando (met bekomt een **.class** file met dezelfde naam). De **.class**-file wordt uitgevoerd (gerund) met het **java**-commando. Het grote voordeel van de .class-files is dat zij platform onafhankelijk zijn, en ze dus zowel op Windows, Mac-OS als Linux draaien. Binnen de multimediawereld draaien de .class files ook op Set-Top boxen voor digitale TV, Blu-Ray Disc spelers, GSM's, PDA's enz...

Oefening:
Download een recente JDK van java.sun.com, en compileer en run hiermee het bovenstaande "EersteProg.java"-programma.

Controle

.....

1.4 Numerieke Variabelen

In VB.NET worden variabelen gedeclareerd met behulp van het Dim-keyword:

```
Dim a As Integer
```

In C en in Java bestaat dit keyword niet, en schrijft men simpelweg:

```
int a;
```

Concreet houdt dit in: reserveer een geheugenplaats voor één variabele van het type integer (dit kan in C 16 bits zijn of 32 bits zijn) afhankelijk van de processor en compiler. In Java echter, is er strict afgesproken hoeveel bits elk type bevat. Voor een **int** in Java is dit **32 bits.**
Op soortgelijke manier kan men in Java andere types van variabelen declareren:

byte a;	8 bits geheel getal, signed, 2's complement genoteerd in het geheugen.
short a;	16 bits geheel getal, signed, 2's complement genoteerd in het geheugen.
int a;	32 bits geheel getal, signed, 2's complement genoteerd in het geheugen.
long a;	64 bits geheel getal, signed, 2's complement genoteerd.

Verder kon je in C aangeven of een int unsigned of signed was, in Java niet. **In Java zijn alle integer-types (byte, short, int, long) altijd signed (2's complement genoteerd).**
Voor komma getallen kan je de volgende types gebruiken (zowel in C als in Java):

float a;	IEEE-754 genoteerd vlottende kommagetal, 32 bits.
double a;	IEEE-754 genoteerd vlottende kommagetal, 64 bits.

In C werden variabelen niet automatisch geïnitialiseerd. Indien je dus niet expliciet opgaf dat een variabele geïnitialiseerd moest worden, werd de waarde van de variabele hetgeen er dan toevallig op de geheugenplaats stond die aan de variabele werd toegewezen.
In Java echter worden de variabelen automatisch op 0 gezet.
Volgend programma geeft het eenvoudig gebruik van variabelen weer:

```
public class EersteProg {
    public static void main(String args[])
    {
            int a;
            float b;
            a=5;
            a=a+1;
            b=3.3;
            b=b-0.1;
    }
}
```

Numerieke variabele kunnen in Java (net als in C) vergeleken worden met behulp van volgende operatoren:

Java (en C) notatie	VB.NET notatie	betekenis
==	=	is gelijk aan
!=	<>	is verschillend van
<	<	is kleiner dan
>	>	is groter dan
<=	<=	is kleiner of gelijk aan
>=	>=	is groter of gelijk aan

Het resultaat van zo'n vergelijking is altijd van het type **boolean**.

1.5 Boolean variabelen

In tegenstelling tot C kent Java echte variabelen van het type boolean die de waarde "true" of "false" kunnen aannemen (in C werden hiervoor integers gebruikt die 0 of 1 werden.)
Dit houdt dan ook in dat je niet kan "rekenen" met variabelen van het type boolean zoals dat in C kon: je kan twee variabelen van het type boolean bijvoorbeeld niet optellen. De volgende LOGISCHE operatoren zijn in Java wel te gebruiken met variabelen van het type boolean:

C notatie	VB.NET notatie	betekenis
==	=	is gelijk aan
!=	<>	is verschillend van
&	AND	logische AND
|	OR	logische OR
!	NOT	logische NOT

1.6 Operatoren

Voor elementaire berekeningen kan men volgende operatoren gebruiken: (de werking is identiek aan de werking in C.)

+	optellen
-	aftrekken
*	vermenigvuldigen
/	delen
%	rest bij deling door

Volgend codefragment geeft een voorbeeld van het gebruik van operatoren:

```
int a,b,c;
a=100;
b=200;
c=a+b; // c wordt 300

int x,y,z;
x= 8;
y=20;
z=y % x;   // z wordt 4
```

1.7 Verkorte schrijfwijze voor operaties

In Java (net als in C) kunnen sommige bewerkingen verkort genoteerd worden. Volgend codefragment geeft hier een voorbeeld van:

```
int i;
i=0;
i++; // i=i+1;
i--; // i=i-1;
i+=7; // i=i+7;
i*=5; // i=i*5;
```

1.8 Het "bereik" (scope) van variabelen

Belangrijk is op te merken dat het "bereik" van een variabele beperkt is. De regel in Java (net als in C) is dat een variabele blijft bestaan totdat de accolade waarbinnen de variabele aangemaakt is, gesloten wordt, daarna wordt de variabele vernietigd, en wordt ze ook niet meer herkend.

1.9 Notatie van getallen

Gehele getallen kunnen in verschillende talstelsels genoteerd worden:

a=512;	decimale notatie
a=0x200;	hexadecimale notatie
a=0345;	octale notatie (begint met 0)

Belangrijk is dat Java octale notatie toestaat (grondtal=8), en dat alle getallen die met 0 beginnen als octale notatie beschouwd worden.
Indien het om een groot getal gaat dat niet binnen het type "int" past dan moet men achter het getal een "L" noteren om aan te geven dat het getal van het type "long" is. Bv. a=13443443443245492L;
Kommagetallen worden best expliciet met een punt genoteerd, ook wanneer er geen mantissa is. Zo zal in Java het getal 53 als een "int" gezien worden, en het getal 53.0 als een "float" gezien worden.

1.10 Karakters

Een karakter in C was één ASCII-teken, en werd met het type “char” aangeduid. In Java heet het type ook “char”, maar gebruikt men **Unicode** in plaats van ASCII-codes. Dit houdt in dat er 16 bits voorzien worden voor het aanduiden van een karakter. Dit levert voldoende mogelijkheden om de vele karakters en vele talen (o.a. arabisch, chinees, thais) te huisvesten. In C was het zo dat je een variabele van het type char ook als getal kon gebruiken, je kon zo bijvoorbeeld twee chars optellen. Je kan in Java nog altijd variabelen van het type “char” gebruiken in rekenkundige uitdrukkingen, maar er is zijn Java minder redenen om dit ook zo te doen.
Net zoals in C wordt een karakter in Java tussen **enkele aanhalingstekens** genoteerd en een string tussen dubbele aanhalingstekens.

```
char a;
a='R';
```

Een Unicode-karakter-code bestaat in Java uit 16 bits, het type “char” heeft dus ook 16 bits (in Java).

```
char b;
b=65;    //de Unicode van 'A'
System.out.println(b); //drukt de letter 'A' af.
```

1.11 Kommentaar

Kommentaar wordt kan in Java (net als in C) op twee manieren genoteerd worden: enerzijds tussen “/*” en “*/”, de kommentaar kan dan over meerdere regels lopen. Indien de kommentaar beperkt is tot één regel dan kan de notatie “//” gebruikt worden.

```
int a; /* Dit is kommentaar, die
          gespreid is over verschillende regels */

int b; // Dit is kommentaar op één regel.
```

Buiten de twee “klassieke” manieren van commentaar is er in Java ook de Javadoc-commentaar. Hiermee kan op basis van de commentaar automatisch documentatie over de programmacode gegenereerd worden. Dit type van commentaar wordt ingeleid door /** en afgesloten met */.
Volgend voorbeeld illustreert het gebruik:

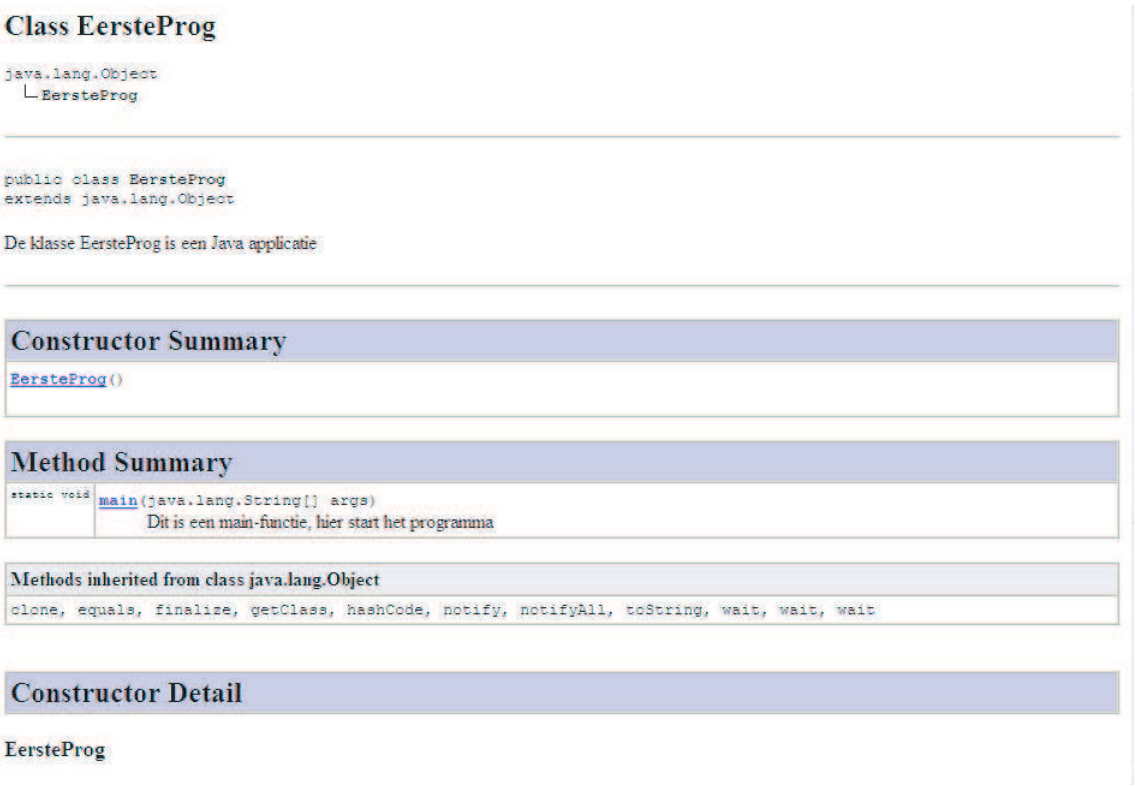
```
import java.lang.*;

/**
 * De klasse EersteProg is een Java applicatie
 *
 * @author       Koen Soontjens
```

```
 * @version      1,5
 */

public class EersteProg {

/**
 * Dit is een main-functie, hier start het programma
 * @param  args   Dit is een array van strings waarmee parameters
 kunnen meegegeven worden vanaf de commandline.
 */
   public static void main(String args[])
   {
       System.out.println("Dit is mijn eerste Java programma\n");
   }
}
```

Op basis van de programmacode kan nu met behulp van javadoc.exe automatisch HTML documentatie gemaakt worden:

```
C:\Program Files\Java\jdk1.6.0_07\bin>javadoc.exe EersteProg.java
Loading source file EersteProg.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_07
Building tree for all the packages and classes...
Generating EersteProg.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
```

De gegenereerde documentatie ziet er zo uit:



Class EersteProg

java.lang.Object
└ EersteProg

public class EersteProg
extends java.lang.Object

De klasse EersteProg is een Java applicatie

Constructor Summary

EersteProg()

Method Summary

static void	main(java.lang.String[] args) Dit is een main-functie, hier start het programma

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

EersteProg

Oefening:
Voorzien volgende code van Javadoc commentaar (de klasse en de twee methodes)

```
import java.lang.*;

 public class EersteProg {

    public static void main(String args[])
    {
    drukaf(100);
    }

    private static void drukaf(int m)
    {
        int a;
        for (a=0;a<m;a++)
            {
                System.out.println(a);
            }

    }
}
```

Controle

.....

1.12 if en else

Met behulp van de operatoren uit vorige paragraaf kan men if-clausules maken. De algemene vorm ziet er zo uit:

```
if ( voorwaarde )
    {
    // uit te voeren als voorwaarde waar is.
    }
else
    {
    // uit te voeren als de voorwaarde niet waar is.
    }
```

Volgend voorbeeld controleert of i gelijk is aan 100:

```
if (i==100)
    {
    System.out.println("i is gelijk aan 100.\n");
    }
else
    {
    System.out.println("i is niet gelijk aan 100.\n");
    }
```

Uiteraard kan de “if”-clausule ook voorkomen zonder “else”-gedeelte.

1.13 Lussen en iteraties

1.13.1 while-lus

De while-lus in Java kent (net als in C) volgende vorm:

```
while ( voorwaarde ) {
    // uit te voeren code
}
```

De code tussen de accolades wordt uitgevoerd zolang de voorwaarde waar is. In onderstaand voorbeeld wordt de lus uitgevoerd zolang i kleiner is dan 100.

```
while ( i < 100 ) {
    i++;
}
```

Merk op dat de lus helemaal niet uitgevoerd wordt als bij aanvang niet aan de voorwaarde (in dit geval i<100) voldaan is.

1.13.2 do while-lus

De do while-lus onderscheidt zich van de while-lus door het feit dat de lus altijd minstens één keer uitgevoerd wordt. De voorwaarde wordt pas op het einde van de lus gecontroleerd, en indien voldaan wordt aan de voorwaarde wordt de lus opnieuw gestart.

```
do {
    // uit te voeren code
} while ( voorwaarde );
```

Onderstaande code geeft een voorbeeld van de do while-lus weer:

```
do {
    i++;
} while ( i<100 );
```

De lus wordt nu mistens één keer uitgevoerd ook wanneer i groter is dan 100 bij aanvang van de lus.

1.13.3 for-lus

De for-lus volgt volgende vorm:

```
for ( initialisatie ; voorwaarde ; wijziging variabele ) {
    //uit te voeren code
}
```

Volgend codefragment geeft een voorbeeld van een eenvoudige for-lus:

```
int i;
for ( i=0 ; i<100 ; i++ ) {
    System.out.println("%d",i); // druk het getal i af.
}
```

Soms is het niet meteen duidelijk tot welke waarde de lus nu uitgevoerd wordt, in dit geval wordt de lus nog uitgevoerd voor het geval i=99, maar niet meer voor het geval i=100. 99 is dus het laatste getal dat afgedrukt wordt. Om twijfel uit te sluiten is het belangrijk in te zien dat de for-lus naar een while-lus omgezet kan worden.

```
initialisatie;
while ( voorwaarde ) {
    // uit te voeren code
    wijziging variabele;
}
```

Onderstaande code geeft het equivalent van bovenstaande for-lus:

```
i=0;
while ( i<100 ) {
    printf("%d",i); // druk het getal i af.
    i++;
}
```

Door de conversie van for- naar while-lus is eenvoudig in te zien dat het laatste getal dat afgedrukt wordt, 99 is. Let wel dat NA het beëindigen van de lus, de waarde van "i" 100 is.

Oefening:
Zet onderstaande for-lus om naar een while lus:

```
for (i=55;i>34;i--)
{
System.out.println(i);
}
```

Controle

.....

1.14 Bitoperaties

Op variabelen van een integer type kunnen (net als in C) volgende operatoren toegepast worden:

&	Bitgewijs en (AND)
|	Bitgewijs en (OR)
^	Bitgewijs en (XOR)
~	Bitgewijs en (NOT)
<<	naar links schuiven
>>	naar rechts schuiven

De term “bitgewijs” duidt aan dat de bewerking in kwestie betrekking heeft op alle paren van bits op dezelfde positie. Met andere woorden, de functie (bv. AND) wordt bit per bit toegepast. Volgend voorbeeld kan hier duidelijkheid in scheppen:

```
200          0000000011001000
341          0000000101010101

200 & 341 = 0000000001000000  =  64
```

Telkens wanneer er in beide getallen op dezelfde postitie een bit op één staat, zal in het resultaat ook de respectievelijke bit op één gezet worden.
Voor de OR operator kan eenzelfde redenering gemaakt worden:

```
200          0000000011001000
341          0000000101010101

200 | 341 = 0000000111011101  =  477
```

De NOT-operator (heeft slechts één getal als input) inverteert elke bit van het getal:

```
 200          0000000011001000
~200          1111111100110111
```

De shiftoperatoren “>>” en “<<” schuiven de bits in een getal respectievelijk een aantal posities naar rechts en naar links:

```
 200          0000000011001000
 200 >> 2     0000000000110010   = 50


 341          0000000101010101
 341 << 3     0000101010101000   = 2728
```

Oefening:
Voer onderstaande "berekening" uit in een Java programma. Verklaar het resultaat (denk aan 2's complement notatie.)

```
System.out.println(~10);
```

Controle

.....

1.15 Primitieve en niet-primitieve types

In Java wordt een duidelijk onderscheid gemaakt tussen primitieve types zoals ints,floats,chars, enz... en niet-primitieve types zoals objecten en arrays.
De primitieve types worden in volgende tabel weergegeven:

byte	short
int	long
float	double
boolean	char

De niet-primitieve types zijn objecten en arrays. **Een belangrijk verschil is dat primitieve types bij methodeaanroepen altijd "by value" doorgegeven worden, en niet-primitieve types altijd "by reference."**

1.16 Arrays

Arrays in Java vertonen duidelijke verschillen met arrays in C. In C gebruikt men de volgende notatie:

```
int a[100];
```

In Java gebeurt het aanmaken van een array in twee delen:

```
int a[]=new int[100];
```

Eerst heeft men in Java de declaratie (int a[]), hiermee geeft je aan dat a een array van int's is, maar je reserveert hiervoor nog geen geheugen, je zegt ook nog niet hoe groot de array moet zijn. Daarna volgt het reserveren van het geheugen voor de array (new int[100]). Hiermee wordt er plaats voor 100 int's gereserveerd. Vermits zoals gezegd in vorige paragraaf een array een niet-primitief type is en het dus bij functieaanroepen doorgegeven wordt "by reference" noemt men het ook een ´´reference type." Men zegt ook wel eens dat a een reference is naar een array van 100 elementen.
Dit maakt het ook mogelijk de reference "a" te koppelen aan een andere array:

```
int a[];
a=new int[100]; //koppel a aan een array
                //(reserveer geheugen voor 100 ints)

a=new int[200]; //koppel a aan een nieuwe array (200 ints, geen
                //behoud van inhoud!)
```

In bovenstaande voorbeeld wordt de reference "a" eerst gekoppeld aan een array van 100 ints, daarna aan een **andere** array van 200 ints. De eerst array van 100 ints wordt vernietigd, de inhoud blijft dus niet behouden in de nieuwe array van 200 ints.
Je kan in Java ook statische arrays maken (de grootte van de array wordt dan automatisch berekend):

```
int a[]={45,346,54,65,112};
```

Meerdimentionale arrays kunnen alsvolgt gemaakt worden:

```
int a[][]=new int[10][100];
```

Deze meerdimentionale arrays worden (net als in C) beschouwd als arrays van arrays.
Een verbetering tenopzicht van C, is dat de lengte van de array als volgt kan opgevraagd worden:

```
int a[]=new int[10];
System.out.println(a.length); //geeft het aantal elementen
```

1.17 Oefeningen

1. Schrijf een programma dat de negen tafels (1 tot en met 9) van vermenigvuldiging weergeeft, in de stijl van 1x1=1, 1x2=2 ... 9x1=9, 9x2=18, 9x9=81.

2. Schrijf een programma dat de dagen van de maand februari in 2009 afdrukt, de eerste dag van de maand is zondag. (zondag 1 februari, maandag 2 februari, ...)

3. Bereken het getal π door volgende som te berekenen. Gebruik hiervoor exact 10000 termen (tussen de haakjes).

$$\pi = 4.(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...)$$

4. Bewaar het getal 4302 in een integer. Bereken het negatief van dit getal zonder de min-operator te gebruiken. Inverteer hiervoor alle bits, en tel er één bij.

5. Bepaal alle priemgetallen kleiner dan 100. Ga hiervoor van elk getal vanaf 3 tot en met 99 na of er geen delers van het getal zijn tussen 1 en het getal-1.

   Controle

   .....

6. Beschouw onderstaande array. Zoek met behulp van een for-lus het grootste element uit de array.

   ```
   int a[8]={12,34,56,78,123,234,99,88};
   ```

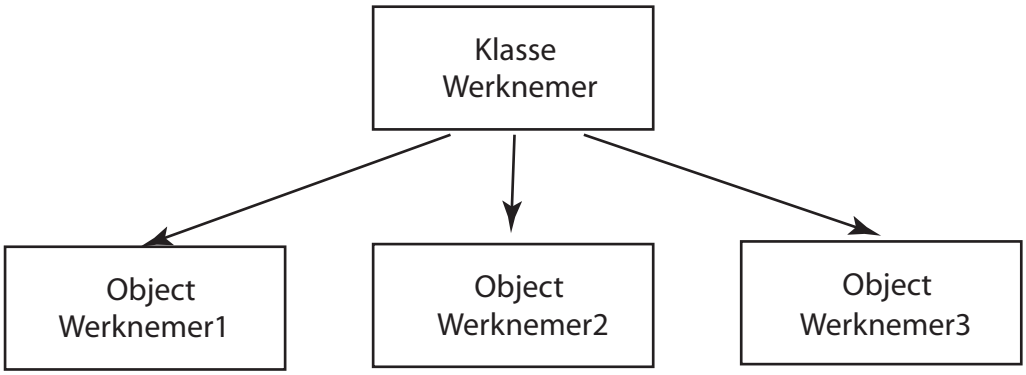
7. Beschouw de array uit vorige oefening. Maak een tweede array b, die je (door telkens het grootste element uit "a" te zoeken) sorteert. Om een getal uit "a" te "verwijderen", kan je het getal op 0 zetten zodat alle andere getallen groter zijn.

   Controle

   .....

1.18 Opbouw van een eenvoudige klasse

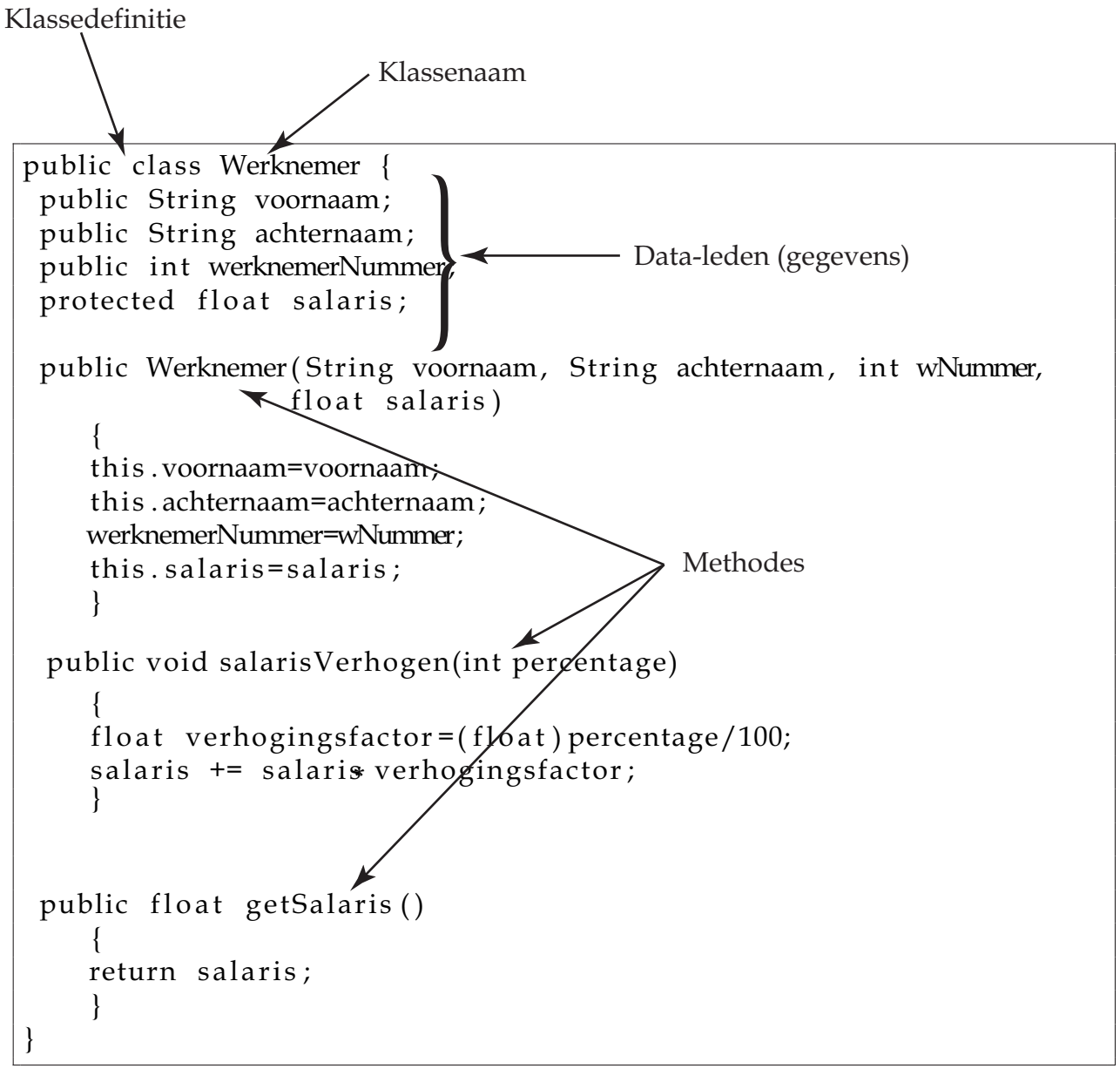
Klassen zijn de fundamentele bouwblokken in Java. Alle gegevens en functionaliteiten worden in Java in klassen georganiseerd. Klassen zijn in feite sjablonen voor objecten. Je kan dus van één klasse meerdere objecten maken, waarbij elk object verschillende gegevens kan bevatten.



De objecten (die volgens het sjabloon van een klasse gemaakt worden) bevatten **gegevens** en **methoden**. De gegevens kunnen uit primitieve types en uit andere objecten bestaan. De methodes zijn functies die toepasbaar zijn op de gegevens van het object.
Een klasse (waarvan deze objecten gemaakt kunnen worden) kan opgesplitst worden in 3 delen:

1. Een klasse declaratie
2. De gegevens of data-leden
3. De methodes (functies)

Onderstaande code geeft een voorbeeld van een klasse-definitie weer:



Bovenaan de klasse vinden we de klasse-definitie:

```
public class Werknemer
```

Deze bevat de klassenaam, in dit geval “Werknemer”. Het is gebruikelijk in Java om de klassenaam met een hoofdletter te schrijven.
Verder geeft het sleutelwoord “public” aan, dat het om een publieke klasse gaat. Dit houdt in dat de klasse zichtbaar (en te gebruiken is) voor alle andere klassen. **Belangrijk is op te merken dat elk bestand in Java slechts één publieke klasse mag bevatten en dat de bestandsnaam dezelfde moet zijn als de naam van de publieke klassenaam.**

Daarna volgen de data-leden van de klasse:

```
public String voornaam;
public String achternaam;
public int werknemerNummer;
protected float salaris;
```

In dit geval werden er vier data-leden gedefinieerd: twee hiervan zijn primitieve types, de andere twee zijn objecten van een andere klasse (de klasse String.)
Elke data-lid declaratie bestaat uit 3 delen:

1. Eerst hebben we de access-modifier, in dit geval public, dit houdt in dat de variabele (het data-lid) voor iedereen toegankelijk is. De variabele salaris heeft de access-modifier "protected": dit houdt in dat de variabele enkel voor methodes van de klasse zelf, afgeleide klassen of klasse uit hetzelfde package (zie later) toegangelijk is.

2. Daarna hebben we het type van de variabele (data-lid) dat zowel een primitief type kan zijn (int,float, enz...) als een niet-primitief type zoals andere klasses of arrays.

3. Tot slot hebben we de naam van het data-lid. Het is in Java gebruikelijk om de variabelenaam met een kleine letter te beginnen, en de daaropvolgende woorden met een hoofdletter te schrijven (zie werknemerNummer).

Vervolgens komen de methodes van de klasse. Deze klasse heeft drie methodes:

```
public Werknemer(String voornaam, String achternaam, int
                    wNummer, float salaris)

public void salarisVerhogen(int percentage)

public float getSalaris()
```

Net als data-leden- en klassedeclaratie wordt de methodedeclaratie gestart met een access-modifier. In dit geval hebben de drie methodes de "public" access-modifier. Hierdoor zijn de methodes van overal aan te roepen.
Eén van de methodes heeft dezelfde naam als de klasse. Methodes met dezelfde naam als de klasse noemt men constructoren. Constructoren worden uitgevoerd telkens men een nieuw object van de klasse maakt (met de new operator).
Elke methode (behalve de constructor) heeft een return type: zo heeft de methode getSalaris als return type "float". Indien men wenst dat de methode geen informatie teruggeeft, gebruikt met het sleutelwoord "void" als return type (zoals bij salarisVerhogen). **Merk op dat constructoren geen return type hebben: zelfs geen "void".**
Indien men geen enkele constructor voorziet in een klasse, zorgt de Java compiler zelf voor een constructor. Deze constructor doet buiten het instantiëren van het object verder niets.

1.19 Objecten van een klasse maken en gebruiken

Herinner u dat alle klassen tot de niet-primitieve types horen. Net als arrays gebeurt het aanmaken van een object dus in twee delen: een eerste deel maakt het duidelijk dat “herman” een reference naar een object van de klasse werknemer wordt:

```
Werknemer herman;
```

Tot hiertoe is er echter nog geen object van het type Werknemer gemaakt. In een tweede stap wordt met behulp van de “new” operator geheugenruimte gereserveerd voor de variabelen van het Werknemer-object (voornaam, achternaam, werknemerNummer en salaris) en wordt de reference “herman” aan deze geheugenruimte gekoppeld. Vanaf dan bestaat het object echt.

```
herman=new Werknemer("Herman", "Hermans",1,1000.0);
```

Merk op dat hierbij de constructor-methode aangeroepen wordt. Deze constructor (zie code) vult de hier meegegeven parameters in, in de data-leden van het Werknemer object. Vanzelfsprekend kan de constructor ook andere taken vervullen indien hij hiervoor geprogrammeerd wordt.
De functie van de new operator is dus tweevoudig: enerzijds wordt er geheugenruimte gereserveerd voor het object, anderzijds wordt de constructor aangeroepen.
Vaak wordt in Java de declaratie en instantiëring op één regel uitgevoerd:

```
Werknemer herman = new Werknemer("Herman", "Hermans",1,1000.0);
```

Nu het object aangemaakt is, kan men het in gebruik nemen. Zowel data-leden als methoden worden (van buiten de klasse) benaderd met behulp van de .-operator. Om bijvoorbeeld het werknemernummer van het object “herman” te benaderen gebruikt men volgende notatie:

```
System.out.println( herman.werknemerNummer );
```

herman.werknemerNummer is dus het data-lid ´´werknemerNummer” dat bij het object “herman” hoort.
Het aanroepen van een methode op het object gebeurt op dezelfde manier:

```
herman.salarisVerhogen( 10 );
```

Bovenstaand statement roept de methode “salarisVerhogen” aan op het object “herman”. Dit houdt in dat de methode “salarisVerhogen” zal werken met de data-leden van het object herman. Deze methode zal het data-lid “salaris” van het object herman verhogen met 10%.
Een ander voorbeeld gebruikt de methode “getSalaris()” om het salaris van een Werknemer-object op te vragen. (Je kan het salaris niet als data-lid opvragen omdat het een “protected” data-lid is, en het dus niet vanuit alle klassen toegankelijk is, de methode getSalaris() is echter public, deze methode geeft een “float” terug die gelijk is aan het data-lid “salaris”)

```
System.out.println( getSalaris() );
```

1.20 Packages

Java klassen worden georganiseerd in “packages”. Packages bevatten doorgaans klassen die samenhoren. Theoretisch zit elke Java klasse in een package. Bovenaan elk .java-bestand wordt het package statement vermeld, gevolgd door de volledig naam van het package:

```
package bedrijf.bibliotheek
```

Met dit statement behoren alle klassen uit dat bestand tot het package “bedrijf.bibliotheek”. Net zoals in Java de bestandsnaam van een .java-bestand overeen komt met de klassenaam van de publieke klasse in het bestand, komt de packagenaam verplicht overeen met de directory-structuur waarin de .java-bestanden zich bevinden. Zo moeten de .java-bestanden van alle klassen die zich in het “bedrijf.bibliotheek”-package bevinden in de map “bedrijf\bibliotheek\” zitten.
De package naam bepaalt mee de volledige naam van een klasse. Stel dat de klasse “Werknemer” in het package “bedrijf.personeelsbeleid” zit, dan is de volledige naam van de klasse “Werknemer”: “bedrijf.personeelsbeleid.Werknemer”.
Zoals reeds vermeld kan je vanuit een andere klasse gebruik maken van de verkorte schrijfwijze “Werknemer” als je bovenaan je code een import zet:

```
import bedrijf.personeelsbeleid.*;
```

Hiermee wordt het mogelijk alle klassen uit het package “bedrijf.personeelsbeleid” verkort te noteren. Je kan ook één enkele klasse uit een package importeren:

```
import bedrijf.personeelsbeleid.Werknemer;
```

1.21 Het “this” sleutelwoord

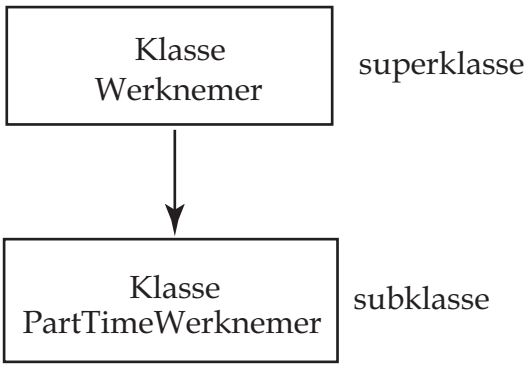
Het “this” sleutelwoord wordt gebruikt om vanuit een methode, andere leden (data of methode) van het huidige object te refereren. In vele gevallen kan het “this” sleutelwoord weggelaten worden, en kunnen leden van het huidige object gewoon met de methode- of variabelenaam benaderd worden. Eén van de meest voorkomende gevallen waarin het gebruik van “this” toch nodig is, is in het geval dat de data-leden overschaduwd worden door een methode-variabele:

```
class Klasse1
    {
    public int a;

    void methode1(int a)
        {
```

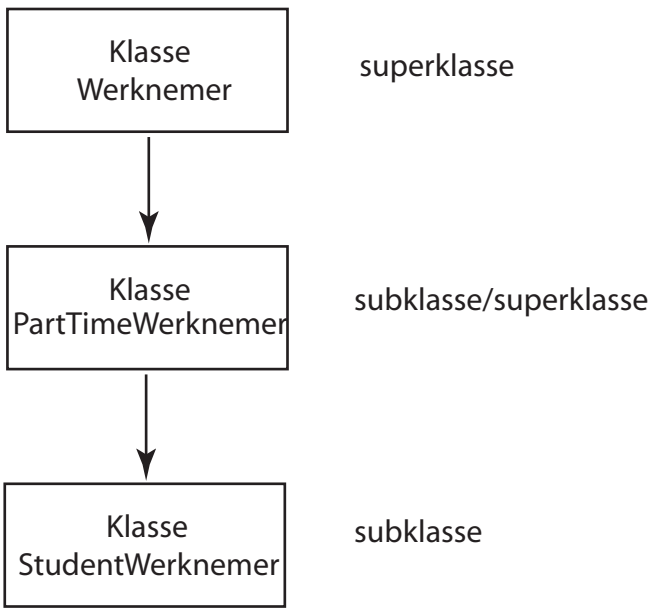
```
        this.a=a;  // hier is het gebruik van this nodig
        }
    }
```

1.22 Afgeleide klassen

Je kan in Java een klasse maken die de data-leden en methodes van een andere klasse overerft. Zo zou je bijvoorbeeld een klasse PartTimeWerknemer kunnen maken, die overerft van de klasse Werknemer. De klasse waarvan je overerft noem je dan de superklasse. De overervende klasse heet de subklasse.



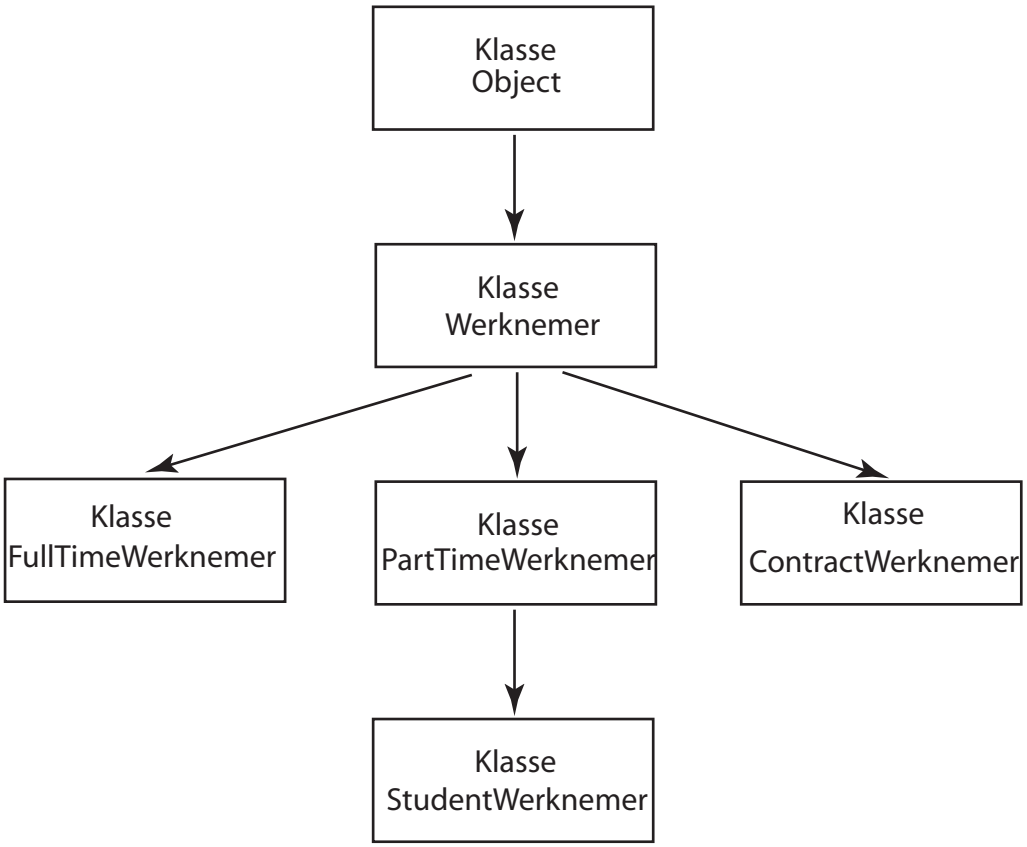
Je kan eventueel op haar beurt van de subklasse overerven. Zo zou men bijvoorbeeld van PartTimeWerknemer een nieuwe subklasse StudentWerknemer kunnen afleiden. Een klasse kan dus tegelijkertijd subklasse zijn van de ene klasse en superklasse zijn van een andere klasse. Deze termen zijn dus relatief.



Om de overerving aan te geven kan je bij de declaratie twee sleutelwoorden gebruiken:

- extends
- implements

Met het sleutelwoord extends geef je bij de declaratie van de subklasse de superklasse op. Op deze manier erft de subklasse alle data-leden en methodes van de opgegeven superklasse (en daarmee dus ook van de superklassen van deze superklasse.)
Als je het sleutelwoord extends niet gebruikt, dan erft jouw klasse toch nog over van de klasse "Object". Daaruit kan je besluiten dat wanneer je van een andere klasse overerft, je ook van Object overerft, omdat de basisklasse van je overervingsketen van Object overerft.



Op die manier hebben alle klassen de klasse "Object" op een of andere manier als superklasse.
In tegenstelling tot C++ ondersteunt Java geen meervoudige overerving: je kan in Java slechts van één klasse overerven. Deze "tekortkoming" wordt in Java opgelost met behulp van interfaces (zie later.)
Laten we even uitgaan van het volgend scenario: PartTimeWerknemers werken

geen volledige week, en hun salaris wordt per uur berekend. Om hun weeksalaris te berekenen maken we gebruik van een nieuw data-lid urenGewerkt, dat vermenigvuldigd met het uurloon (salaris) het weeksalaris opleverd.

```
public class PartTimeWerknemer extends Werknemer
{

 public int urenGewerkt;

 public PartTimeWerknemer ( String voornaam, String achternaam,
                                int nr, float sal, int urengw)
    {
    super(voornaam, achternaam, nr, sal);
    this.urenGewerkt=urengw;
    }

 public float getWeekLoon ()
    {
    return this.salaris * (float)this.urenGewerkt;
    }
 }
```

Bovenaan de code wordt aangegeven dat je een nieuwe klasse PartTimeWerknemer wil maken, die overerft van de klasse Werknemer, en die dus alle data-leden en methoden van Werknemer bevat:

```
public class PartTimeWerknemer extends Werknemer
```

De nieuwe klasse heeft automatisch alle data-leden van Werknemer, maar ook één data-lid extra (tov. Werknemer):

```
public int urenGewerkt;
```

Dit data-lid bevat het aantal uren dat de PartTimeWerknemer gewerkt heeft per week.

Daarnaast is er een nieuwe constructor voorzien voor PartTimeWerknemer: constructoren gedragen zich wat betreft overerving niet zoals gewone methodes, ze worden niet mee overgeërfd (zie paragraaf “Constructoren en overerving.”)

```
 public PartTimeWerknemer ( String voornaam, String achternaam,
                                int nr, float sal, int urengw)
    {
    super(voornaam, achternaam, nr, sal);
    this.urenGewerkt=urengw;
    }
```

De nieuwe constructor voert op de eerste regel de constructor van de superklasse (Werknemer) uit met het statement:

```
super(voornaam, achternaam, nr, sal);
```

Daarna wordt het data-lid “urenGewerkt” ingevuld met de extra constructorparameter “urengw”:

```
this.urenGewerkt=urengw;
```

Hoewel de afgeleide subklasse “PartTimeWerknemer” alle methoden van “Werknemer” overerft, kan je nieuwe methoden gaan toevoegen aan de subklasse. We kunnen drie manieren onderscheiden:

- **Overschrijven (Overriding)**: hierbij wordt een methode met dezelfde naam en dezelfde parameters als in de superklasse gedefinieerd. De methode vervangt dan de methode van de superklasse voor alle objecten van de subklasse. Op deze manier zou je in de klasse “PartTimeWerknemer” de methode “salarisVerhogen” kunnen overschrijven:

```
    public void salarisVerhogen(int percentage)
    {
        if (percentage>5)
        {
        percentage=0;
        System.out.println("Fout: slechts
                            5% opslag toegestaan");
        }
        else
        {
        super.salarisVerhogen(percentage);
        }
    }
```

  De nieuwe methode zorgt ervoor dat “PartTimeWerknemers” niet meer dan 5% opslag kunnen krijgen. Merk op dat je bij het overschrijven van een methode de “oude” methode (van Werknemer) kunt aanroepen door gebruik te maken van het sleutelwoord “super”.

- **Overloading**: hierbij wordt een methode met dezelfde naam, maar met verschillende parameters als in de superklasse gedefinieerd. De methode bestaat dan naast de methode van de superklasse, en de combinatie van de parameters bepaalt welke methode aangeroepen wordt. Zo zou men een tweede methode “getSalaris(char muntEenheid)” kunnen definiëren, die samen met de methode “getSalaris()” van de superklasse te gebruiken is op objecten van de klasse “PartTimeWerknemer:”

```
float getSalaris (char muntEenheid)
{
    if (muntEenheid=='$')
      return (salaris/0.68F); //salaris in $
    else
      return (salaris);      // salaris in Euro
}
```

  De tweede methode “getSalaris” kan men aanroepen op objecten van het type “PartTimeWerknemer” met een karakter als parameters, maar men kan

ook nog altijd de methode “getSalaris()” gebruiken zonder parameters (uit de Werknemer-klasse).

- **Toevoegen**: Daarnaast is het ook nog mogelijk om gewoon “extra” methoden voor de afgeleide klasse te voorzien, met een andere naam als de methoden uit de superklasse. Een voorbeeld is de methode “getWeekloon()”, deze bestond nog niet in de klasse Werknemer:

```
public float getWeekLoon ()
    {
    return this.salaris * (float)this.urenGewerkt;
    }
```

1.23 Constructoren en overerving

Constructoren gedragen zich bij overerving opmerkelijk anders dan gewone methodes. Eerst en vooral worden ze niet zoals gewone methodes mee overgeërfd. Verder is het belangrijk in te zien dat constructoren voornamelijk gebruikt worden om het object te initialiseren. Om een object correct te initialiseren moet niet alleen de constructor van de huidige klasse uitgevoerd worden, maar ook alle constructoren van de superklassen. Daarom is men verplicht om op de eerste regel van de nieuwe constructor de constructor van de superklasse aan te roepen met behulp van het super-keyword. Indien men niet expliciet opgeeft op de eerste regel van de constructor dat men met behulp van “super” de consturctor van de superklasse wil op roepen, roept de compiler automatisch de default-constructor op van de superklasse (dit is de constructor zonder parameters).
De verplichting om “super” op de eerste regel van de nieuwe constructor aan te roepen heeft tot gevolg dat bij het aanroepen van een constructor van een overervende klasse, eerst de constructor van de basisklasse “Object” (dit is de basisklasse van alle objecten) wordt uitgevoerd, daarna wordt de constructor van de klasse uitgevoerd die rechtstreeks overerft van “Object”, vervolgens wordt de constructor van de volgende subklasse uitgevoerd, tot uiteindelijk de constructor van de te instantiëren (met het new-keyword) klasse aangeroepen wordt.

1.24 Abstracte klassen en methoden

De definitie van een abstracte methode is eenvoudig: het is een methode die gedeclareerd wordt, maar zonder implementatie blijft:

```
abstract class Klasse1
    {
    abstract void methode1(int a); //methode zonder implementatie
    }
```

Een abstracte methode wordt aangeduid met het keyword “abstract”.
Een abstracte klasse is een klasse waarvan één of meerdere methoden abstract zijn. Er kunnen geen objecten geïnstantieerd worden van abstracte klassen (met behulp van de new operator). Men kan wel van abstracte klassen overerven, en in de afgeleidde klasse een implementatie voorzien voor de abstracte methoden van de superklasse.

1.25 Interfaces

Het concept “interfaces” komt tegemoed aan het gebrek aan meervoudige overerving. Zo kan een klasse (eventueel bovenop het gebruik van “extends”) één of meerdere interfaces “implementeren” met het sleutelwoord “implements.”
Een interface is een soort abstracte klasse waarbij alle methoden abstract zijn, die gedeclareerd wordt met het sleutelwoord “interface” ipv. “class”:

```
public interface Kleurbaar
    {
    public abstract void geefKleur(Color C); // abstracte methode
                                             // (zonder implementatie)
    }
```

Een andere klasse kan dan de interface implementeren met behulp van het “implements” sleutelwoord. Hiermee verplicht de klasse zich ertoe een implementatie te voorzien voor alle methoden van de interface.

```
public class Cirkel implements Kleurbaar extends Figuur {
    public void geefKleur(Color C)
    {
        // hier volgt de implementatie van
        // geefKleur voor objecten van het type Cirkel
    }
}
```

In bovenstaande declaratie erft “Cirkel” van “Figuur” over, en implementeerd “Cirkel” de interface Kleurbaar. Met andere woorden wordt er een implementatie voorzien voor de methode “geefKleur(Color C).”

1.26 Access modifiers

Access modifiers bepalen of objecten van andere klassen bepaalde data-leden of methoden kunnen gebruiken. Dit kan op twee niveaus:

- Op het niveau van de klasse definitie, kan men **public** of “package-private” (**geen modifier**) gebruiken.
- Op het niveau van de leden (data-lid of methode) kan men **public**, **protected**, **private** of “package-private” (**geen modifier**) gebruiken.

Een klasse kan public gemaakt worden waardoor ze zichbaar is voor alle andere klassen. Indien bij de klassedeclaratie geen access-modifier opgegeven wordt, spreekt men van “package-private” en is de klasse enkel zichtbaar voor klassen binnen hetzelfde package.
Op het niveau van de leden kan men public of “package-private” (geen modifier) gebruiken, met dezelfde betekenis als bij klassen. Maar men kan ook private of protected gebruiken. De **private** modifier zorgt ervoor dat het lid enkel zichtbaar is van binnen de klasse. De **protected** modifier doet hetzelfde, maar maakt het lid ook zichtbaar voor afgeleide klassen en klassen uit hetzelfde package.

1.27 Oefeningen

1. Definieer de klasse “Werknemer” zoals beschreven in de cursus in het bestand “Werknemer.java”. Maak in je hoofdprogramma 4 objecten aan van het type “Werknemer”.

2. Verhoog het salaris van de eerste twee werknemers met 10%. Druk de salarissen van de vier werknemers af.

3. Definieer de klasse “PartTimeWerknemer” zoals beschreven in de cursus in het bestand “PartTimeWerknemer.java”. Maak in je hoofdprogramma 2 objecten van het type “PartTimeWerknemer.”

4. Probeer het salaris van de eerste “PartTimeWerknemer” te verhogen met 10%. Overschrijf de methode “salarisVerhogen” om de maximale opslag te beperken tot 5% (zoals beschreven in de cursus.) Probeer opnieuw het salaris te verhogen tot 10%. Druk de salarissen van de twee werknemers af.

5. Voeg aan de klasse “Werknemer” een data-lid “RSZpercentage” (float, private) toe dat bepaalt hoeveel RSZ de werkgever betaald voor deze werknemer. Stel de waarde standaard in op 33%. Voeg ook de methode “setRSZ(float RSZ)” toe, die het RSZ percentage invult, en de methode “getRSZ()”, die het RSZ percentage uitleest.

6. Probeer de nieuwe methodes uit op een object van het type “Werknemer”, probeer het daarna uit op een object van het type “PartTimeWerknemer”.

7. Maak een nieuwe klasse “StudentWerknemer” die overerft van “PartTimeWerknemer”. Stel voor de nieuwe klasse de variabele RSZpercentage standaard in op 5%. Maak een nieuw object van “StudentWerknemer” en controleer het RSZ percentage.

8. Maak een nieuwe interface “Betaalbaar” die een methode “void betaal()” definieert.

9. Zorg ervoor dat de klasse “Werknemer” de interface “Betaalbaar” implementeert. Implementeer de methode “betaal” in de klasse “Werknemer”. De methode moet de tekst afdrukken: “Betaal het salaris van … aan de werknemer …”, waarbij het salaris en de naam van de werknemer ingevuld wordt. Test de methode “betaal()” op objecten van “Werknemer”, “PartTimeWerknemer” en “StudentWerknemer”.

10. Maak een nieuwe klasse “Faktuur”, met als public data-leden “faktuurNr” en “faktuurBedrag”, die eveneens de “Betaalbaar”-interface implementeerd. Implementeer de “betaal”-methode. De methode moet volgende tekst afdrukken: “Betaal het faktuur … voor een bedrag van …”. Maak een object van “Faktuur”.

Hoofdstuk 2

MHP

2.1 Mijn eerste Xlet

Een MHP programma wordt een Xlet genoemd, en wordt niet zoals een klassiek Java programma met een main-methode gestart. Een Xlet lijkt veel meer op een Java Applet, dat zich in verschillende toestanden kan bevinden.
Om een Xlet te maken moet men een class maken die de javax.tv.xlet.Xlet interface implementeerd. De Xlet interface bestaat heeft 4 methodes, die je als implementeerde klasse moet invullen:

- initXlet()
- startXlet()
- pauseXlet()
- destroyXlet()

De Xlet kent dus 4 toestanden:
Voor het initialiseren gebruikt de Xlet de initXlet() methode, die geïmplementeerd moet worden. Deze functie wordt uitgevoerd vóór het starten van de Xlet. Na de initialisering wordt de methode startXlet() aangeroepen. Hier pas mag men beginnen met grafische methodes aan te roepen en te reageren op input van de gebruiker. De derde toestand voorziet erin dat de Xlet in een rusttoestand geplaatst kan worden waarbij de niet-benodigde resources vrijgegeven kunnen worden. Deze methode noemt pauseXlet(). In de vierde toestand zorgt de methode destroyXlet() ervoor, dat de Xlet afgesloten wordt en alle gebruikte resources vrijgegeven worden.
Door de vier methodes te implementeren, bekomt met een eenvoudige Xlet.
Het is belangrijk bij de implementatie, om alleen initialisering en reservatie van resources te doen in de initXlet() methode, en pas in de startXlet() methode over te gaan tot concrete probleemoplossing.
De methode pauseXlet() en destroyXlet worden pas echt belangrijk op het moment dat meerdere Xlets tegelijk lopen.

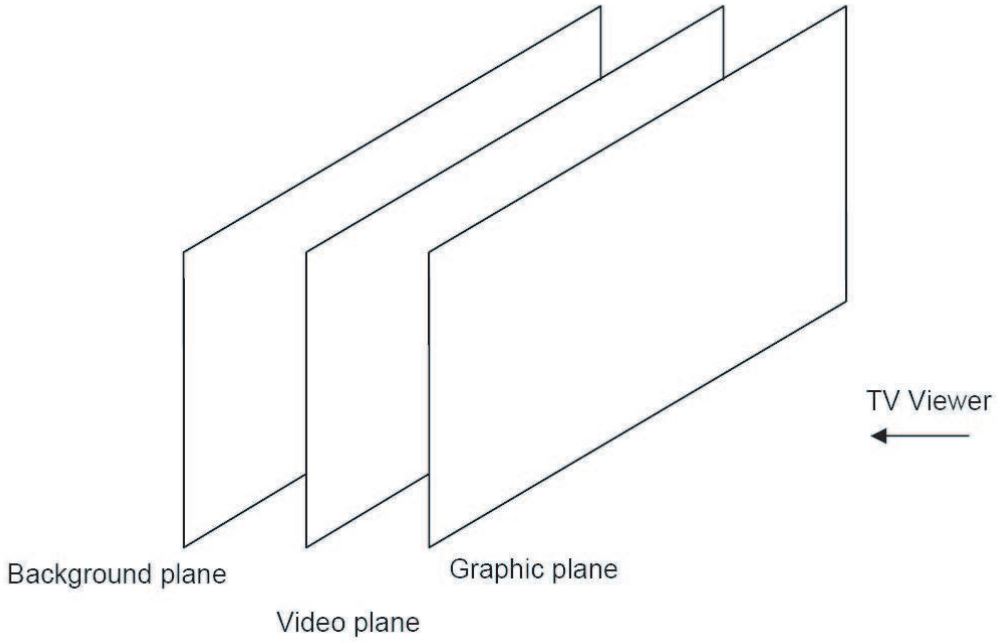
```
// Om javax.tv.xlet.Xlet verkort als Xlet te
// schrijven importeren we javax.tv.xlet.*

import javax.tv.xlet.*;

public class MijnEersteXlet implements Xlet {
    // Variabele om de actuele Xlet-context in te bewaren.
    private XletContext actueleXletContext;


// Initialiseren van de benodigde resources en variabelen:
    public void initXlet(XletContext context)
    {
        this.actueleXletContext = context;
    }

// Starten van de Xlet:
    public void startXlet() throws XletStateChangeException
    {
        // Communicatie (In- en Uitvoer met de gebruiker)
    }

// Methode voor de pause toestand.
    public void pauseXlet()
    {
        // vrijgeven van niet-nodige resources
    }

// Beëindigen van de Xlet.
    public void destroyXlet(boolean unconditional)
                                throws XletStateChangeException
    {
        if (unconditional) {
        // System.out.println geeft debug in weer voor emulatoren.
            System.out.println("De Xlet moet beëindigd worden");
        }
        else {
            System.out.println("De mogelijkheid bestaat "+
            "door het werpen van een exceptie "
            +"de Xlet in leven te houden. ");
            throw new XletStateChangeException("Laat me leven!");
        }
    }
}
```

2.2 Grafische Xlet

Het grafisch model van MHP bestaat uit drie lagen:

- Het background plane kan een kleur of MPEG I-frame weergeven.

- Het video plane kan MPEG-2/4 video weergeven.
- Het graphic plane kan verschillende Scenes bevatten die op hun beurt Text, Buttons en Images kunnen bevatten.



Voor het graphics plane heeft MHP gekozen om de klassen van havi.ui te gebruiken. Deze bevatten naast een subset uit java.awt speciale componenten voor multimedia apparaten zoals set-top boxen. Om iets te tekenen heb je allereerst iets nodig om op te tekenen. In ons geval is dit een "Scene". Op de Scene kunnen we kant-en-klare componenten zetten zoals Buttons of Images, of we kunnen het als leeg blad gebruiken en er zelf via basisfuncties op tekenen. Je kan de twee manieren ook door elkaar gebruiken.
Een Scene is een object van de klasse org.havi.ui.HScene. Om een Scene object te bekomen, moet je je gewenste eigenschappen van de Scene aan een HSceneFactory doorgeven met behulp van een HSceneTemplate object. En de HSceneFactory geeft je dan een Scene object dat zoveel mogelijk aan deze wensen voldoet. Zo bevat de HSceneTemplate hoe groot je de Scene wenst.

```
import org.havi.ui.*;

...

HScene scene;

// Het template maken
HSceneTemplate sceneTemplate = new HSceneTemplate();
```

```
// Grootte en positie ingeven
sceneTemplate.setPreference(
org.havi.ui.HSceneTemplate.SCENE_SCREEN_DIMENSION,
            new HScreenDimension(1.0f, 1.0f),org.havi.ui.HSceneTemplate.
                REQUIRED);

sceneTemplate.setPreference(org.havi.ui.HSceneTemplate.SCENE_SCREEN_LOCATION,
            new HScreenPoint(0.0f, 0.0f), org.havi.ui.HSceneTemplate.REQUIRED)
                ;

// Een instantie van een Scene vragen aan de factory
scene = HSceneFactory.getInstance().getBestScene(sceneTemplate);


// Scene zichtbaar maken
scene.validate(); scene.setVisible(true);
```

De HScreenDimension wordt uitgedrukt als percentage van de totale schermgrootte. Indien je (0.5f,0.5f) doorgeeft, betekent die een vierde van de schermoppervlakte. De HScreenPoint geeft de oorsprong aan, waarbij (0.0f,0.0f) links boven is en (0.5f, 0.5f) in het midden van het scherm is.
Bij het gebruik van een Scene moet men er rekening mee houden dat de aspect ratio verschillend is van een PC, en afhankelijk van het beeldformaat (4:3, 14:9 of 16:9).
In het voorbeeld van PAL (720 x 576) moet men in het achterhoofd houden dat de aspectratio geen 1:1 is zoals bij een PC, maar 1:1,067 en dat men van zichtbaar bereik op het scherm ongeveer 5% aan de rand verliest.
Verder is het aan te raden rekening te houden met de interlacing. Dit houdt in dat sterke contrastverschillen tussen 2 pixels verticaal aanleiding kunnen geven te flikkeringen.

2.3 Kleurgebruik in MHP

Specifiek aan grafische objecten in MHP is de mogelijkheid tot het gebruik van transparante kleuren. Zo kan zowel de video/I-frame zichtbaar gemaakt worden als de bovenliggende grafische objecten.
De klasse DVBColor maakt het mogelijk om deze transparante kleuren te definiëren.
De MHP 1.02 Standaard ondersteunt slechts een beperkt aantal kleuren, en op de eerste generatie MHP-set-top-boxen kunnen alleen de volgende kleuren gebruikt worden. Pas met nieuwere MHP-standaarden wordt een uitgebreider pallet ondersteund.

transparantie	alpha	grijs (r=g=b)	rood	groen	blauw
0%	255	42, 85, 170, 212	0, 63,127, 191, 255	0, 31, 63, 95, 127, 159, 191, 223, 255	0, 127, 255
30%	179	–	0, 85, 170, 255	0, 51, 102, 153, 204, 255	0, 255

2.4 Grafische Componenten

Eens we beschikken over een Scene kunnen we aan deze scene objecten (Widgets) toevoegen. In MHP bestaan hiervoor de buttons, tekstvelden, invulboxen en radiobuttons uit het org.havi.ui package. Let op: men gebruikt in MHP niet de componenten uit java.awt zoals in klassieke Java programma's. Volgende voorbeeld toont hoe je een HStaticText aan de Scene toevoegd:

```
package hellotvxlet;

import javax.tv.xlet.*;
import org.dvb.ui.*;
import java.awt.*;

//Stap 1: klassen onder org.have.ui verkort noteren
import org.havi.ui.*;


public class HelloTVXlet implements Xlet {

    private XletContext actueleXletContext;
    private HScene scene;
    // debuggen activeren of niet ?
    private boolean debug=true;

    // Stap 2: tekstLabel declareren
    private HStaticText tekstLabel;

    public void initXlet(XletContext context) throws XletStateChangeException
    {
        if(debug) System.out.println("Xlet Initialiseren");
        this.actueleXletContext = context;
        // Template aanmaken
        HSceneTemplate sceneTemplate = new HSceneTemplate();

        // Grootte en positie aangeven
  sceneTemplate.setPreference(HSceneTemplate.SCENE_SCREEN_DIMENSION,
  new HScreenDimension(1.0f, 1.0f), HSceneTemplate.REQUIRED);

        sceneTemplate.setPreference(HSceneTemplate.SCENE_SCREEN_LOCATION,
        new HScreenPoint(0.0f, 0.0f), HSceneTemplate.REQUIRED);
```

```
        // Een instantie van de Scene aanvragen aan de factory.
        scene = HSceneFactory.getInstance().getBestScene(sceneTemplate);



        // Stap 3: object aanmaken
            tekstLabel = new HStaticText("TEKST");

        // Stap 4: eigenschappen van tekstLabel instellen
            tekstLabel.setLocation(100,100);
            tekstLabel.setSize(400,250);
            tekstLabel.setBackground(new DVBColor(255,255,255,179));
            tekstLabel.setBackgroundMode(HVisible.BACKGROUND_FILL);

        //Stap 5: tekstLabel aan de Scene toevoegen.
        scene.add(tekstLabel);
    }

    public void startXlet() throws XletStateChangeException
    {

        if(debug) System.out.println("Xlet Starten");
        //Scene zichtbaar maken

        scene.validate();
        scene.setVisible(true);
    }

    public void pauseXlet()
    {

    }

    public void destroyXlet(boolean unconditional) throws
       XletStateChangeException
    {

    }

}
```

Er zijn 5 stappen nodig:

1. Eerst wordt bovenaan de code aangegeven dat men de klassen onder “org.havi.ui” wil importeren. We willen immers de klasse HStaticText gebruiken, die voluit org.havi.ui.HStaticText heet.

2. Daarna maakt men een nieuw data-lid “tekstLabel” genaamd. Merk op dat hierbij de new-operator nog niet gebruikt wordt, er is dus in feite nog geen object, enkel een reference naar een object van de klasse HStaticText, maar de reference “tekstLabel” zal doorheen gans de klasse gekend zijn.

3. In de methode initXlet wordt met behulp van de new-operator een nieuw object gemaakt, en wordt dit object aan de reference “tekstLabel” gekoppeld.

4. Daarna worden met behulp van de methoden setLocation, setSize, setBackground, setBackgroundMode van de HStaticText-klasse de eigenschappen van het object ingesteld.

5. Tot slot wordt het HStaticText object aan de scene toegevoegd.

Naast elementen van de klasse “HStaticText” bevat de HAVI-bibliotheek (Home Audio/Video Interoperability, org.havi) tal van andere klassen, zoals bijvoorbeeld “HTextButton”. Deze klasse erft over van HStaticText, en ondersteunt dus ook de methoden van “HStaticText”, maar is bovendien “Actionable”. Je kan met andere woorden een knop van de klasse “HTextButton” de focus geven, en erop klikken (met de “OK” toest van de afstandsbediening). Meestal zal de knop ook bij een andere toestand (gefocuseerd of aangeklikt) een andere layout vertonen. In de Javadoc van MHP[1] vind je nog veel meer elementen uit de HAVI-bibliotheek die je op een scene kan plaatsen.
Volgend codefragment laat zien hoe je met bijna identiek dezelfde code als voor een HStaticText een HTextButton kan toevoegen aan de scene.

```
package hellotvxlet;

import javax.tv.xlet.*;
import org.dvb.ui.*;
import java.awt.*;


import org.havi.ui.*;


public class HelloTVXlet implements Xlet {

    private XletContext actueleXletContext;
    private HScene scene;
    // debuggen activeren of niet ?
    private boolean debug=true;

    private HTextButton knop1,knop2;

    public void initXlet(XletContext context) throws XletStateChangeException
    {
        if(debug) System.out.println("Xlet Initialiseren");
        this.actueleXletContext = context;
        // Template aanmaken
        HSceneTemplate sceneTemplate = new HSceneTemplate();

        // Grootte en positie aangeven
  sceneTemplate.setPreference(HSceneTemplate.SCENE_SCREEN_DIMENSION,
  new HScreenDimension(1.0f, 1.0f), HSceneTemplate.REQUIRED);
```

[1]Op de server geïnstalleerd onder C:\javadoc

```
        sceneTemplate.setPreference(HSceneTemplate.SCENE_SCREEN_LOCATION,
        new HScreenPoint(0.0f, 0.0f), HSceneTemplate.REQUIRED);


        // Een instantie van de Scene aanvragen aan de factory.
        scene = HSceneFactory.getInstance().getBestScene(sceneTemplate);

            knop1 = new HTextButton("KNOP1");
            knop1.setLocation(100,100);
            knop1.setSize(100,50);
            knop1.setBackground(new DVBColor(0,0,0,179));
            knop1.setBackgroundMode(HVisible.BACKGROUND_FILL);

        scene.add(knop1);


            knop2 = new HTextButton("KNOP2");
            knop2.setLocation(100,200);
            knop2.setSize(100,50);
            knop2.setBackground(new DVBColor(0,0,0,179));
            knop2.setBackgroundMode(HVisible.BACKGROUND_FILL);

        scene.add(knop2);
    }

    public void startXlet() throws XletStateChangeException
    {

        if(debug) System.out.println("Xlet Starten");
        //Scene zichtbaar maken

        scene.validate();
        scene.setVisible(true);
    }

    public void pauseXlet()
    {

    }

    public void destroyXlet(boolean unconditional) throws
       XletStateChangeException
    {

    }

}
```

Uiteindelijk heeft het gebruik van knoppen in plaats van tekstlabels maar zin wanneer de knoppen navigeerbaar zijn (je kan met de pijltjes een knop selecteren) en wanneer de knoppen aanklikbaar zijn (je kan met “OK” een knop “aanklikken”).

Volgend voorbeeld laat zien hoe je die kan implementeren:

```
            knop1 = new HTextButton("KNOP1");
            knop1.setLocation(100,100);
            knop1.setSize(100,50);
            knop1.setBackground(new DVBColor(0,0,0,179));
            knop1.setBackgroundMode(HVisible.BACKGROUND_FILL);

            knop2 = new HTextButton("KNOP2");
            knop2.setLocation(100,200);
            knop2.setSize(100,50);
            knop2.setBackground(new DVBColor(0,0,0,179));
            knop2.setBackgroundMode(HVisible.BACKGROUND_FILL);

        knop1.setFocusTraversal(null, knop2, null, null); // op, neer, links,
            rechts
        knop2.setFocusTraversal(knop1, null, null, null); //  op, neer, links,
             rechts

        scene.add(knop1);
        scene.add(knop2);

        knop1.requestFocus();
```

De methode “setFocusTravesal” kan toegepast worden op alle objecten van klassen die navigeerbaar zijn (die de interface HNavigable implementeren), naast HTextbutton zijn dit ook: (HAnimation, HIcon, HText, HRange, HGraphicButton HTextButton, HToggleButton, HListGroup, HSinglelineEntry HMultilineEntry en HRangeValue). Met de methode setFocusTraversal stel je in naar welk element (HTextButton of ander element dat HNavigable implemeneteerd) je navigeert op het moment dat je boven, onder, links of rechts drukt op je afstandsbediening. **Merk op dat de methode “requestFocus()” op één van de knoppen aangeroepen moet worden NADAT de knoppen aan de scene zijn toegevoegd.**
Om nu acties te koppelen aan knoppen kent men met behulp van setActionCommand een unieke string to aan elke knop:

```
knop1.setActionCommand("knop1_actioned");
knop2.setActionCommand("knop2_actioned");
```

Daarna geeft men met behulp van “addHActionListener” aan naar welk object de string gestuurd moeten worden, wanneer men op de knop “klikt:”

```
knop1.addHActionListener(this);
knop2.addHActionListener(this);
```

In dit geval is ervoor gekozen om de strings te laten versturen naar de Xlet zelf, dit kan echter evengoed een ander object zijn.
Het ontvangende object (in dit geval this) moet de “HActionListener” interface implementeren:

```
public class HelloTVXlet implements Xlet, HActionListener
```

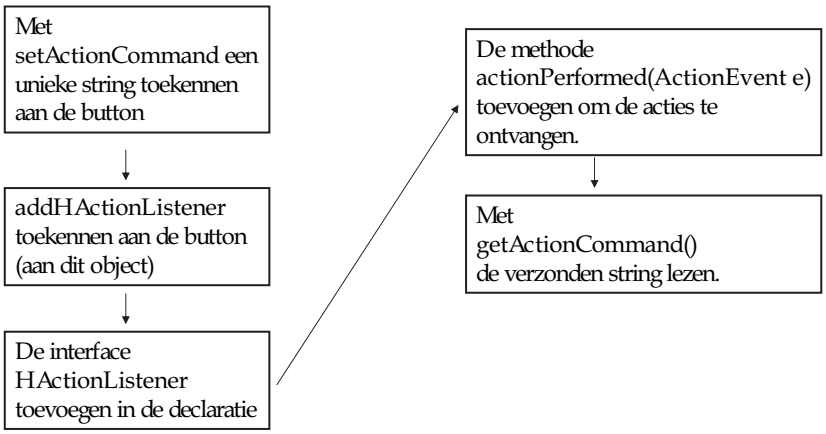
Dit houdt in dat het object de methode “actionPerformed(ActionEvent e)” moet implementeren:

```
public void actionPerformed(ActionEvent e)
    {
      System.out.println(e.getActionCommand());
    }
```

Aan de hand van de binnengekomen strings kan men (eventueel met behulp van een switch-case statement) op een correcte manier reageren op het aanklikken van de knoppen.
Voor dit alles heeft men volgende imports nodig:

```
import java.awt.event.*;
import org.havi.ui.event.*;
```

Schematisch kan men het proces als volgt samenvatten:



2.5 Oefeningen

1. Bouw met behulp van HStaticText en HTextButton een “Wie wordt multimiljonair?”-spel, waarbij de gebruiker met behulp van de pijltjes kan nagiveren tussen 4 antwoorden, en één antwoord kan kiezen. De Xlet geeft na het selecteren van het antwoord aan of het juist of fout was.

2. Voeg een extra knop “Hulplijn aanvragen” toe, die twee (foutieve) van de vier antwoorden weghaalt.

2.6 Tekenen en zelfgemaakte objecten

2.6.1 Nieuwe componenten maken

Naast kant en klare klassen zoals HStaticText kan men zelf ook componenten maken. Hiertoe maakt men een nieuwe klasse die van de klasse “HComponent” overerft. In deze klasse kan men de “paint (Graphics g)” methode overschrijven. Het MHP-framework zal dan jouw paint-methode aanroepen op het moment dat de component getekend moet worden. Met behulp van de graphische context “Graphics g” kan je lijnen, figuren en teksten tekenen. Volgend voorbeeld laat zien hoe je een nieuwe component maakt:

```
package hellotvxlet;


import org.havi.ui.*;
import java.awt.*;
import org.dvb.ui.*;

// Klasse van Hcomponent overerven
public class MijnComponent extends HComponent  {

    //Plaats en locatie instellen in de constructor
    public MijnComponent()
    {
    this.setBounds(0, 0, 100,100);
    }

    // Paint methode overschrijven
    public void paint(Graphics g)
    {
        g.setColor(new DVBColor(100,100,100,179));
        g.fillRect(0, 0, 100 , 100);
        g.setColor(Color.white);
        g.drawString("Tekst1",15,20);
    }
}
```

Merk op dat in de constructor de (in dit geval vaste) plaats en grootte van de component wordt ingesteld met de methode “this.setBounds”, dit is een methode uit de HComponent-klasse.

De nieuwe component kan men dan net als een HStaticText op de scene plaatsen:

```
MijnComponent mc=new MijnComponent();
scene.add(mc);
```

2.6.2 Oefeningen

1. Pas de constructor van MijnComponent aan zodat men bij het aanmaken van het object de positie en grootte kan meegeven.

2. Teken (in een MijnComponent) een blauw, transparant, afgeronde rechthoek, met daarom in niet compatibele transparante gele kleur een tekst. Gebruik enkel MHP 1.02 kleuren.

3. Voeg aan de vorige rechthoek een schaduweffect toe.

2.7 Images weergeven

Bitmaps kunnen op de klassieke Java manier worden ingevoegd, met die beperking dat enkel kleuren binnen het MHP palet weergegeven kunnen worden. Men begint best door een klasse (bv. MijnComponent) te laten overerven van HComponent en de paint-funtie van de nieuwe klasse te overschrijven. In deze paint-functie gebruik je dan g.drawImage(...); om de bitmap weer te geven. Uiteraard moet je in de Xlet eerst een HScene aanvragen, en een object van je MijnComponent-klasse toe voegen aan de HScene. Het laden van de bitmap kan met één regel geschieden:

```
java.awt.Image imgjpg = this.getToolkit().getImage("bitmap.jpg");
```

De klasse HComponent beschikt over een Toolkit die de functie getImage bevat. Het laden is echter een asynchroon process (het gebeurt in de achtergrond, zonder dat hierop gewacht wordt). Het is daarom aan te raden het laadproces te volgen met behulp van een MediaTracker object, dit om te voorkomen dat de bitmap getekend wordt vooraleer hij geladen is.

```
MediaTracker mtrack=new MediaTracker(this);
mtrack.addImage(bmap,1);
```

De MediaTracker wordt gestart met als parameter de HComponent waarin de bitmaps uiteindelijk getekend zullen worden. Daarna wordt het Image aan de MediaTracker toegevoegd met de methode addImage(..., id). Waarbij id het referentienummer is. Het laden en weergeven van images wordt geïllustreerd in onderstaande code:

```
package hellotvxlet;


import org.havi.ui.*;
import java.awt.*;


// Klasse van Hcomponent overerven
public class MijnComponent extends HComponent  {

    private Image bmap;
```

```
    private MediaTracker mtrack;

    //Plaats en locatie instellen in de constructor
    public MijnComponent(String bitmapnaam, int x, int y)
    {

    bmap=this.getToolkit().getImage(bitmapnaam);
    mtrack=new MediaTracker(this);
    mtrack.addImage(bmap, 0);
    try
    {
    mtrack.waitForAll();              // wacht tot alle bitmaps geladen zijn
    }
    catch (Exception e)
        {
        System.out.println(e.toString());
        }
    this.setBounds(x, y, bmap.getWidth(null),bmap.getWidth(null));
      // opgegeven plaats en afmeting van de bitmap
    }
    // Paint methode overriden
    public void paint(Graphics g)
    {
        g.drawImage(bmap, 0, 0, null);
    }
}
```

2.8 Gebruikersinvoer

Gebruikersinvoer kan ruwweg op twee manier gebeuren:

1. Met behulp van UserEventListener
2. Met behulp van KeyEventListener

Aangezien de UserEventListener-manier het meeste mogelijkheden biedt, wordt hier enkel deze methode besproken.

De events worden intern als volgt genoteerd:

VK_UP	Pijl omhoog	not standardized
VK_DOWN	Pijl omlaag	not standardized
VK_LEFT	Pijl links	not standardized
VK_RIGHT	Pijl rechts	not standardized
VK_ENTER	Enter	not standardized
VK_0 - VK_9	Cijfer 1-9	48-57
VK_TELETEXT	Teletexttoets	459
VK_COLORED_KEY_0	Rood	403
VK_COLORED_KEY_1	Geel	404
VK_COLORED_KEY_2	Groen	405
VK_COLORED_KEY_3	Blauw	406

Met behulp van een UserEventRepository object kunnen events waarop de Xlet moet reageren vastgelegd worden. Met de methode addUserEventListener() van het EventManager() object is het dan mogelijk om de UserEventListener te registreren. Door de methodes van de interface te implementeren kan men dan op events reageren:

```
package hellotvxlet;

import javax.tv.xlet.*;
import org.havi.ui.*;
import java.awt.event.*;
import org.havi.ui.event.*;
import org.dvb.event.*;

public class HelloTVXlet implements Xlet, UserEventListener {

    public void initXlet(XletContext xletContext) throws
       XletStateChangeException
    {

    }

    public void startXlet() throws XletStateChangeException
    {
        System.out.println("Xlet gestart");
        // EventManager aanvragen
        EventManager manager = EventManager.getInstance();

        // Repository
        UserEventRepository repository = new UserEventRepository( "Voorbeeld"
           );

        // Events toevoegen
        repository.addKey( org.havi.ui.event.HRcEvent.VK_UP );
        repository.addKey( org.havi.ui.event.HRcEvent.VK_DOWN );

        // Bekend maken bij EventManager
        manager.addUserEventListener( this, repository);
    }
```

```
    public void pauseXlet() {
    }

    public void destroyXlet(boolean unconditional) throws
        XletStateChangeException {
    }

    // Opvangen van de Key Events
    public void userEventReceived(org.dvb.event.UserEvent e) {
        if(e.getType() == KeyEvent.KEY_PRESSED){
            System.out.println("Pushed Button");
            switch( e.getCode() ) {
                case HRcEvent.VK_UP:
                    System.out.println("VK_UP is PRESSED");
                    break;
                case HRcEvent.VK_DOWN:
                    System.out.println("VK_DOWN is PRESSED");
                    break;
            }
        }
    }

}
```

2.9 Oefeningen

Schrijf een Xlet die met behulp van de pijltjes op de afstandsbediening een bitmap kan verplaatsen op het scherm.

2.10 Timers

De tot hiertoe beschreven programmas voeren slechts acties uit op commando van gebruikersinvoer. Om continu acties uit te voeren (bijvoorbeeld elke 5000 ms) kan men gebruik maken van timers. Eerst en vooral maakt men een aparte klasse die overerft van java.util.TimerTask. Hierin is men verplicht een run() methode te hebben die de acties uitvoert op de ingestelde tijdstippen:

```
package hellotvxlet;

import java.util.TimerTask;


public class MijnTimerTask extends TimerTask {

public void run()
    {
    System.out.println("Timer method");
    }
}
```

In het hoofdprogramma maakt men dan een nieuw object van de “MijnTimerTask-klasse.” Vervolgens maakt men een object van de klasse “Timer”, en gebruikt men de methode “scheduleAtFixedRate(MijnTimerTask object, start (ms), herhaal (ms)” om de run()-methode uit de methode “MijnTimerTask-klasse” op vaste tijdsstippen aan te roepen.

```
package hellotvxlet;

import javax.tv.xlet.*;
import org.havi.ui.*;
import java.awt.event.*;
import org.havi.ui.event.*;
import org.dvb.event.*;
import java.util.Timer;



public class HelloTVXlet implements Xlet {

    public void initXlet(XletContext xletContext) throws
       XletStateChangeException
    {

    }

    public void startXlet() throws XletStateChangeException
    {
         MijnTimerTask objMijnTimerTask=new MijnTimerTask();
         Timer timer = new Timer();
         timer.scheduleAtFixedRate(objMijnTimerTask,0 , 5000); //start na 0ms,
              elke 5000ms
    }

    public void pauseXlet() {
    }

    public void destroyXlet(boolean unconditional) throws
       XletStateChangeException {
    }
}
```

2.11 Oefeningen

Schrijf een Xlet die een bitmap (sterrenhemel) laat scrollen over het scherm.

2.12 Achtergronden

Het beperkte kleurenpallet van MHP laat niet toe om full color beelden weer te geven. Een oplossing hiervoor kan gevonden worden in het gebruik van het

"background plane". Een MHP systeem laat toe hier een MPEG I-frame op weer te geven. Dit is te vergelijken met een JPEG bestand, dat in full color weergegeven wordt. Natuurlijk is het niet direct mogelijk om hier dynamische gegevens op weer te geven, tenzij je Java code schrijft die de MPEG I-frame data zelf opbouwt. Veelal zal het MPEG I-frame statische informatie bevatten zoals sfeerbeelden of teksten die niet wijzigen. Om een MPEG I-frame te maken maak je eerst een PNG bestand met je favoriete grafische software (Photoshop, Illustrator, etc...), en zet je vervolgens dit bestand met ffmpeg (freeware) om in een MPEG I-frame. Hiertoe kan je het volgende commando gebruiken:

```
ffmpeg -f image2 -i image.png image.m2v
```

Een MHP settop box toont het I-frame via zijn hardware MPEG-decoder.
Aan het tonen van een I-frame in een Xlet gaan volgende zaken vooraf:
De configuratie van alle layers (background, video en graphics) begint bij het HScreen object. Elke MHP box heeft één HScreen object per fysiek scherm dat er aan aangesloten is. Typisch is dit dus één HScreen object per box. Elk HScreen object heeft een aantal HScreenDevice objecten. Deze stellen de verschillende lagen voor in het scherm. Normaal zal een HScreen object dus drie objecten kennen die elk een instatie zijn van een subklasse van HScreenDevice:

- HBackgroundDevice, die background plane voorsteld.
- HVideoDevice, die de video plane voorsteld.
- HGraphicsDevice, die de graphics plane voorsteld.

Bij het onderzoeken van de Javadoc file over HScreen vinden we drie methodes van HScreen terug die ons de drie HScreenDevice objecten kunnen geven:

```
public HVideoDevice getDefaultHVideoDevice()
public HGraphicsDevice getDefaultHGraphicsDevice()
public HBackgroundDevice getDefaultHBackgroundDevice()
```

Van sommige categorieën van devices (video en graphics) kan je meerdere devices hebben. Bij video denken we dan aan picture-in-picture. Er kan echter slechts één HBackgroundDevice zijn.
Eenmaal we een HScreenDevice hebben van één van de drie categorieën moeten we dit gaan reserveren met de reserveDevice(ResourceClient r) methode. Hiervoor moeten we als parameter een klasse opgeven die ResourceClient implementeerd, en dus de volgende methodes bevat:

```
public void notifyRelease(ResourceProxy proxy) { }
public void release(ResourceProxy proxy) { }
public boolean requestRelease(ResourceProxy proxy, Object requestData) {
return false; }
```

Daarna kunnen we het HScreenDevice gaan configureren. Hiervoor gebruiken we net als bij een HScene een template. In dit geval moet de template een instantie van een subklasse van HScreenConfigTemplate zijn, dus: HBackground-ConfigTemplate, HGraphicsConfigTemplate of HVideoConfigTemplate.

Op basis van het template laten we door het device een configuratie maken. We gebruiken de methode getBestConfiguration waarnaar we het template versturen. Indien dit lukt, gebruiken we de methode setBestConfiguration op het device om de template te activeren.
Volgend voorbeeld illustreert dit:

```
        // Template maken voor Background Device
        bgTemplate = new HBackgroundConfigTemplate();

        // We wensen een "Still Image" achtergrond
        bgTemplate.setPreference(HBackgroundConfigTemplate.STILL_IMAGE,
                    HBackgroundConfigTemplate.REQUIRED);

        // Template opsturen
        bgConfiguration = (HStillImageBackgroundConfiguration)bgDevice.
            getBestConfiguration(bgTemplate);

        try {
        // Configuratie toepassen
            bgDevice.setBackgroundConfiguration(bgConfiguration);
             }
             catch (java.lang.Exception e){
            System.out.println(e.toString());
             }
```

Om uiteindelijk een achtergrond weer te geven, moet de achtergrond eerst geladen worden. De achtergronden zijn objecten van de klasse “HBackgroundImage”, de constructor bevat de bestanden naam:

```
private HBackgroundImage agrondimg=new HBackgroundImage("pizza1.m2v");
```

Om de achtergrond te laden gebruik men de load(HBackgroundImageListener h) methode. Hierbij wordt een object meegegeven dat de HBackgroundImageListener-interface implementeerd en volgende methoden bevat:

```
public void imageLoaded(HBackgroundImageEvent e)
{

}
public void imageLoadFailed(HBackgroundImageEvent e)
{

}
```

Indien het laden (met behulp van de load-methode) lukt zal de methode imageLoaded aangeroepen worden, meestal bevat deze methode code om de achtergrond weer te geven:

```
public void imageLoaded(HBackgroundImageEvent e)
{
 try {
        bgConfiguration.displayImage(agrondimg);
    }
    catch (Exception s){
```

```
    System.out.println(s.toString());
    }
}
```

Onderstaand voorbeeld geeft de volledige code weer om een achtergrond te laden en te tonen:

```
package hellotvxlet;

import javax.tv.xlet.*;
import org.havi.ui.*;
import org.davic.resources.*;
import org.havi.ui.event.*;

public class HelloTVXlet implements Xlet,ResourceClient,
    HBackgroundImageListener
{
    private HScreen screen;
    private HBackgroundDevice bgDevice;
    private HBackgroundConfigTemplate bgTemplate;
    private HStillImageBackgroundConfiguration bgConfiguration;
    private HBackgroundImage agrondimg=new HBackgroundImage("pizza1.m2v");

    public void notifyRelease(ResourceProxy proxy) { }
    public void release(ResourceProxy proxy) { }
    public boolean requestRelease(ResourceProxy proxy, Object requestData) {
        return false; }

public void imageLoaded(HBackgroundImageEvent e)
{
 try {
        bgConfiguration.displayImage(agrondimg);
    }
    catch (Exception s){
  System.out.println(s.toString());
   }
}
public void imageLoadFailed(HBackgroundImageEvent e)
{
    System.out.println("Image kan niet geladen worden.");
}


    public void initXlet(XletContext context)
    {
        // HScreen object opvragen
        screen = HScreen.getDefaultHScreen();

        // HBackgroundDevice opvragen
        bgDevice = screen.getDefaultHBackgroundDevice();

         // HBackgroundDevice proberen te reserveren
         if (bgDevice.reserveDevice(this)) {
             System.out.println("BackgroundImage device has been reserved");
```

```
                } else {
              System.out.println("Background image device cannot be reserved");
              }

        // Template maken
        bgTemplate = new HBackgroundConfigTemplate();

        // Configuratieinstelling "STILL_IMAGE"
        bgTemplate.setPreference(HBackgroundConfigTemplate.STILL_IMAGE,
                  HBackgroundConfigTemplate.REQUIRED);

        // Configuratie aanvragen en activeren indien OK
        bgConfiguration = (HStillImageBackgroundConfiguration)bgDevice.
            getBestConfiguration(bgTemplate);
  try {
            bgDevice.setBackgroundConfiguration(bgConfiguration);
             }
             catch (java.lang.Exception e){
            System.out.println(e.toString());
             }
    }

    public void startXlet()
    {
        System.out.println("startXlet");
            //Image laden
         agrondimg.load(this);
    }

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

    public void destroyXlet(boolean unconditional)
    {
        System.out.println("destroyXlet");
  grondimg.flush();
    }

}
```

2.13 Oefening

1. Schrijf een programma waarbij je vier achtergronden (pizza1.m2v, pizza2.m2v, pizza3.m2v en pizza4.m2v) laat roteren met behulp van de pijltjestoetsen.

2. Pas de code van oefening 1aan met een tekstvak waarin de pizza's worden toegevoegd wanneer de gebruiker op "OK" drukt.