

## Hoofdstuk 5: De Java-programmeertaal

### 5.1 Inleiding

In het vorige hoofdstuk hebben we reeds kennismaking gedaan met programmeeralgoritmen. We hebben onder andere gezien dat we voor gegevens ruimte moeten maken en dat programmeerinstructies elkaar opvolgen en een bepaald pad volgen. Door een beslissing kan dit pad een andere kant opgaan en tevens is het mogelijk dat bepaalde instructies herhaald worden. In een voorbeeld hebben we dit reeds vertaald naar concrete Java-code.

In dit hoofdstuk gaan we de taalregels (*syntax*) van Java systematisch onder de loep nemen. Deze bestaan uit de volgende onderdelen:

1. De variabelen en hun mogelijke waarden.
2. De bewerkingen of operatoren.
3. Uitdrukkingen.
4. Programmeerregels.
5. Code-blokken.
6. Instructies voor het verloop van het programma.

In de volgende paragrafen leren we de taalregels van Java kennen en tegelijkertijd leren we programmeeralgoritmen maken.

### 5.2 Variabelen en letterlijke waarden

In een programmeertaal moeten we gebruik kunnen maken van gegevens. Deze gegevens worden tijdelijk opgeslagen in het geheugen zodat de computer er bewerkingen mee kan doen. Zo'n geheugenplaats krijgt een naam en we noemen dit een 'variabele', omdat de waarde die we in deze geheugenplaats steken variabel is.

Een variabele is dus een gegevenseenheid met een bepaalde naam. Je kan het vergelijken met een doosje met een etiket. In het doosje kan je bepaalde dingen steken maar elk doosje is maar bedoeld voor één soort dingen.

Een variabele heeft de volgende kenmerken:

1. **Type:** In Java is iedere variabele van een bepaald type. Het type van de variabele bepaalt welk soort gegevens de variabele kan bevatten en welke bewerkingen men ermee kan uitvoeren; bijvoorbeeld getallen of een reeks van letters. Het type is als het ware het 'soort' van dingen dat je in het doosje kan steken: bijvoorbeeld een doosje voor suikerklontjes, of een doosje voor ontbijtgranen.
2. **Naam (*identifier*):** Iedere variabele heeft een naam die bestaat uit letters en cijfers. De naam moet steeds beginnen met een letter. Hij komt overeen met het etiket op het doosje.
3. **Bereik (*scope*):** Iedere variabele heeft een bepaald bereik. Dat wil zeggen dat een variabele in een programma maar gebruikt kan worden binnen een bepaald gebied van de programmacode. Doosjes voor de keuken kan je alleen in de keuken gebruiken en niet in de badkamer.

#### 5.2.1 De declaratie van variabelen

Alvorens een variabele te kunnen gebruiken moet hij gedeclareerd worden. We moeten eigenlijk eerst de geheugenruimte in de computer reserveren vooraleer we deze kunnen

gebruiken. Dit gebeurt op de volgende manier:

```
type name;
```

Voorbeeld:

```
int number;
```

We geven dus eerst aan van welk datatype de variabele is en vervolgens wat zijn naam is. Er zal dus in het geheugen plaats gemaakt worden voor een gegeven van een bepaald type en deze geheugenplaats krijgt een naam, zodat we die nadien makkelijk kunnen aanduiden.

Verschillende variabelen van hetzelfde type kunnen op dezelfde regel gedeclareerd worden. De namen van de variabelen worden dan gescheiden door een komma:

```
type name1, name2, name3;
```

Concreet voorbeeld:

```
int number1, number2, number3;
```

Men kan tijdens de declaratie reeds een initiële waarde toekennen aan de variabele. Dit gebeurt op de volgende manier:

```
type name = initialValue;
```

Concreet voorbeeld:

```
int number = 5;
```

Indien meerdere variabelen op dezelfde regel gedeclareerd worden, verloopt de initialisatie als volgt:

```
type name1 = initialValue1, name2 = initialValue2;
```

Concreet voorbeeld:

```
int number1= 3, number2= 7, number3 = 9;
```

Bovenstaande syntax is mogelijk, maar niet aanbevolen, omdat het de code minder goed leesbaar maakt en het opzoeken van de declaratie van een variabele bemoeilijkt.

Het toekennen van een waarde aan een variabele kan ook na de declaratie gebeuren:

```
type name;  
name = value;
```

Voorbeeld:

```
int number;  
number = 5;
```

Voor het toekennen van een initiële waarde kan men gebruikmaken van een *literal* (letterlijke waarde) of een andere variabele.

Voorbeeld:

```
int number1 = 5;           // Initialisation by literal  
int number2 = number1;    // Initialisation by variable
```

## 5.2.2 Het datatype

In Java kan men het datatype onderverdelen in twee grote categorieën: het **primitieve datatype** en het **referentietype**.

### 5.2.2.1 Het primitieve datatype

Het **primitieve datatype** bevat een enkele waarde. Bijvoorbeeld een getal, een letter of een logische (boolaanse) waarde: waar (*true*) – niet waar (*false*).

In de onderstaande tabel zijn alle primitieve datatypes opgesomd met hun minimale en maximale waarde.

Type	Omschrijving	Formaat	Minimaal	Maximaal
<i>Gehele getallen</i>				
byte	Byte waarde	8 bit two's complement	-128	127
short	Short integer	16 bit two's complement	-32768	32767
int	Integer	32 bit two's complement	$-2^{31}$	$2^{31}-1$
long	Long integer	64 bit two's complement	$-2^{63}$	$2^{63}-1$
<i>Reële getallen</i>				
float	Single precision floating point	32 bit IEEE 754	-3.4E+38	3.4E+38
double	Double precision floating point	64 bit IEEE 754	-1.7E+308	1.7E+308
<i>Andere types</i>				
boolean	Boolean waarde	1 bit	false	true
char	Karakter	16 bit Unicode	\u0000	\uFFFF

Tabel 1: Primitieve datatypes

In Java hebben alle getallen een teken. Er bestaan hier geen *unsigned integers* en dergelijke.

In tegenstelling tot sommige andere programmeertalen is het formaat van de datatypes in Java steeds hetzelfde voor elk platform. In C/C++ bijvoorbeeld kan de lengte van een integer variëren naargelang het platform. Bij het ontwikkelen van overdraagbare programma's moet de programmeur hier heel rigoureus rekening houden met de verschillende formaten op de diverse platformen.

Bij Java daarentegen kan de programmeur uitgaan van een vast formaat.

Karakters worden in Java als 16-bits UNICODE-karakters voorgesteld. Men kan karakters ook als 16-bits-getallen (zonder teken) beschouwen waar men bepaalde bewerkingen op kan doen.

Indien de 9 meest beduidende bits gelijk zijn aan 0, komt de UNICODE-codering overeen met de ASCII-codering. ASCII is dus een *subset* van UNICODE.

Voorbeelden van primitieve variabelen:

```
int anIntegerNumber;
float aDecimalNumber;
```

### 5.2.2.2 Het referentietype

Het referentietype is een verwijzing of referentie naar een object. Het referentietype bevat het adres van het object. De inhoud van de variabele is dus eigenlijk de geheugenlocatie van iets anders.

In de volgende code wordt een nieuw object gecreëerd van de klasse *Button*. De variabele *button* is een verwijzing naar dit object. De inhoud van deze variabele is het adres van het object.

```
Button button = new Button();
```



Afbeelding 33: Een referentie naar een object

Een referentievariabele die naar geen enkel object verwijst, heeft de voorgedefinieerde waarde `null`.

In het verder verloop van de cursus zullen we uitvoerig aandacht besteden aan het gebruik van het referentietype. Het is van het uiterste belang het onderscheid tussen het primitieve datatype en het referentietype goed te maken.

### 5.2.2.3 Afleiding van het datatype voor lokale variabelen

Indien we een variabele declareren en onmiddellijk een waarde toekennen is het niet altijd nodig om het datatype expliciet op te geven. Indien de compiler uit de context kan afleiden wat het datatype is, kunnen we ook gebruikmaken van het algemene type `var`.

Bijvoorbeeld:

```
var number = 5;
```

In dit geval kan de compiler uit de toekenning afleiden dat het datatype `int` is omdat de waarde 5 van het type `int` is. Daarom zal ook de variabele `number` van het type `int` zijn maar het volstaat om in dit geval `var` te gebruiken.

De compiler zal in dit geval `var` vervangen door `int`.

Dit afleiden van het datatype kan niet indien de variabele pas later een waarde krijgt. Dit is

daarom niet toegelaten:

```
var number; // Not allowed !!
number = 5;
```

Het afleiden het datatype en het gebruik van `var` is enkel bedoeld om in complexe omstandigheden de hoeveelheid code te reduceren. Het introduceert geenszins een nieuw datatype en zeker geen dynamisch datatype zoals we dat bijvoorbeeld in *JavaScript* kennen.

Deze afleiding van het datatype kan overigens enkel gebruikt worden voor lokale variabelen. We zullen later zien wat zo'n lokale variabele is.

### 5.2.3 Literals

*Literals* zijn letterlijke waarden die men opneemt in het programma: bijvoorbeeld `5`, `'a'`, `true`. Ze worden gebruikt om variabelen te initialiseren, als parameter van een methode of als waarde in een berekening.

Voor ieder primitief datatype bestaan er *literals*.

#### 5.2.3.1 Boolean-literals

De enige boolean-*literals* zijn de gereserveerde woorden `true` en `false`.

```
boolean b1 = true;
boolean b2 = false;
```

Met het afgeleide datatype wordt dit:

```
var b1 = true;
var b2 = false;
```

Het is hier voor de compiler duidelijk dat het type `boolean` is.

#### 5.2.3.2 Karakter-literals

Karakter-*literals* worden voorgesteld door het karakter tussen enkelvoudige aanhalingstekens.

```
char c = 'a';
```

Met het afgeleide type wordt dit:

```
var c = 'a';
```

Indien het karakter niet beschikbaar is op het toetsenbord, kan men ook gebruikmaken van de UNICODE-notatie. Dit is een 16-bits getal in hexadecimale vorm.

```
char c = '\u45FA';
```

De `\u` geeft aan dat het hier om UNICODE gaat.

Vermits karakters niet meer zijn dan een 16-bits index in de UNICODE-tabel kan men ook om het even welk positief 16-bits getal toekennen aan een karakter:

```
char c = 123;
```

Hier kunnen we geen gebruikmaken van het afgeleide type want de compiler ziet 123 op de eerste plaats als een `int`.

Er zijn een aantal *escape-codes* voor speciale karakters:

<b>Escape code</b>	<b>Betekenis</b>
'\n'	Newline: nieuwe regel.
'\r'	Return: wagenterugkeer.
'\t'	Tabulator.
'\b'	Backspace: terugkeertoets.
'\f'	Formfeed: nieuwe pagina.
'\''	Enkelvoudig aanhalingsteken.
'\"'	Dubbel aanhalingsteken.
'\\'	Backslash: schuine streep.

Tabel 2: Escape-codes

Deze tekens worden gebruikt bij het afdrukken van karakters op het scherm of op een printer.

### 5.2.3.3 Literals voor gehele getallen

Getallen worden standaard weergegeven in decimale notatie. Het is echter ook mogelijk de binaire, octale en hexadecimale notatie in *literals* te gebruiken. Voor de binaire notatie zet men `0b` of `0B` voor het getal, voor de octale notatie een `0` (nul) en voor de hexadecimale notatie `0x` of `0X`.

<b>Talstelsel</b>	<b>Notatie</b>	<b>Grondtal</b>	<b>Geldige cijfers</b>
binair	<code>0b10010110</code> <code>0B10010110</code>	2	0 1
octaal	<code>042</code>	8	0 1 2 3 4 5 6 7
decimaal	<code>39</code>	10	0 1 2 3 4 5 6 7 8 9
hexadecimaal	<code>0x5B</code> <code>0X5B</code>	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Tabel 3: Talstelsels voor gehele getallen

Gehele getallen worden als `integer` (32 bit) beschouwd. Om een *literal* als `long` te beschouwen, zet men er de letter `L` achter.

```
long aLong = 152125789458L;
```

Dit is enkel nodig indien de letterlijke waarde groter is dan de maximale `integer`-waarde of indien we het datatype laten afleiden door de compiler:

```
var aLong = 5L;
```

Zonder de toegevoegde `L` zal de compiler het getal 5 als `int` interpreteren.

Om grote getallen beter leesbaar te maken is het toegestaan liggende streepjes (*underscores*) toe te voegen:

```
long value = 152_125_789_458L;
```

Dit kan toegepast worden bij alle getaltypes: `byte`, `short`, `int`, `long`, `float`, `double` en dit voor alle notaties: decimaal, octaal en binair.

Voorbeelden:

```
byte b = 0b0011_1001;
short s = 072_23;
int i = 0xFA_FF_23_AB;
```

De *underscore* mag evenwel niet op volgende plaatsen voorkomen:

- aan het begin of einde van een getal.
- voor de achtervoegsels `L` en `F`.
- grenzend aan de decimale punt.
- op plaatsen waar een getal in de vorm van een tekenreeks (String) verwacht wordt.

#### 5.2.3.4 Floating point literals

De volgende *literals* worden beschouwd als getallen met drijvende komma:

1. Getallen met een decimaal punt: `1.235`.
2. Getallen met wetenschappelijke notatie: `45E+3`.
3. Getallen met de letter `D` achter zijn van het type `double`: `459D`.
4. Getallen met de letter `F` achter zijn van het type `float`: `5687F`.

*Floating-point literals* zonder de letter `D` of `F` zijn standaard van het type `double`.

Indien we het datatype laten afleiden door de compiler is ook hier het gebruik van `D` of `F` aangewezen om het juiste datatype te bekomen:

```
var f = 5F;
var d = 5D;
```

Ook hier kan men gebruikmaken van *underscores*:

```
float f = 215_778.456_458F;
double d = 458_589_125.457_456;
```

#### 5.2.3.5 String literals

Strings zijn tekenreeksen en worden later in de cursus behandeld. Voor de volledigheid vermelden we hier dat string *literals* bestaan uit een reeks karakters tussen dubbele aanhalingstekens.

```
System.out.println("Hello World!");
```

## 5.2.4 De naam

Iedere variabele heeft een naam of *identifier* die aan de volgende voorwaarden moet voldoen:

1. De naam bestaat uit **een reeks Unicode-karakters**<sup>1</sup> beginnend met een letter, met een dollar-teken (\$) of underscore-teken (\_). De naam mag niet bestaan uit enkel het underscore-teken.
2. De naam mag **geen gereserveerd woord** zijn.

Onderstaande tabel geeft de gereserveerde woorden in Java weer:

abstract	boolean	break	byte	case
catch	char	class	const <sup>2</sup>	continue
default	do	double	else	extends
final	finally	float	for	goto <sup>3</sup>
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while	assert	enum

Tabel 4: Gereserveerde woorden in Java

3. De naam moet **uniek** zijn binnen zijn bereik (*scope*). Men mag dus geen twee variabelen binnen hetzelfde bereik dezelfde naam geven.

Het is een gewoonte om variabelen met een kleine letter te laten beginnen terwijl klassen met een grote letter beginnen. Als een variabele uit meerdere woorden bestaat, dan begint ieder woord met een hoofdletter, behalve het eerste woord. Men noemt dit de *CamelCase*-notatie, naar analogie van de bulten van een kameel.

De namen van variabelen zijn doorgaans in het Engels, aangezien dit de voertaal is onder programmeurs.

Voorbeeld:

```
int anInteger;
int aSecondInteger;
```

### Opdracht 1: Data types

In deze opdracht maken we een eenvoudig Java-programma dat de verschillende datatypes gebruikt en afdrukt op het scherm. We definiëren voor elk datatype een lokale variabele in de methode main() van het programma. Om waarden op het scherm af te drukken, gebruiken

1 Meer informatie over de UNICODE standaard is te vinden op de volgende website:  
[www.unicode.org](http://www.unicode.org)

2 Wordt niet gebruikt in Java maar is wel een gereserveerd woord. Om constanten te definiëren gebruikt men *final*.

3 Wordt (gelukkig) niet gebruikt in Java maar is wel een gereserveerd woord.

we de methode `System.out.println()`. We kunnen de variabele die we willen afdrukken gewoon tussen de ronde haken plaatsen.

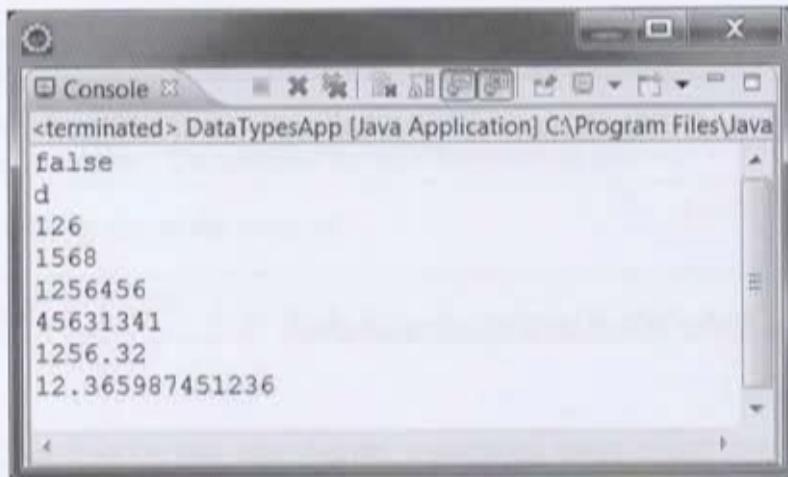
- Tik de volgende code in en schrijf het bestand weg onder de naam `DataTypesApp.java` in de map `src\exercise05_01`.

```
package exercise05_01;

public class DataTypesApp {
    public static void main(String[] args) {
        boolean aBoolean = false;
        char aCharacter = 'd';
        byte aByte = 126;
        short aShortInteger = 1568;
        int anInteger = 1256456;
        long aLongInteger = 45631341L;
        float aDecimalNumber = 1256.32F;
        double aBigDecimalNumber = 12.365987451236;

        System.out.println(aBoolean);
        System.out.println(aCharacter);
        System.out.println(aByte);
        System.out.println(aShortInteger);
        System.out.println(anInteger);
        System.out.println(aLongInteger);
        System.out.println(aDecimalNumber);
        System.out.println(aBigDecimalNumber);
    }
}
```

- Compileer het programma en voer het uit.



- Geef de integer de octale waarde 0342.
- Geef de integer de hexadecimale waarde 0x56\_31.
- Geef de integer de binaire waarde 0b0101\_1100.
- Vervang waar mogelijk de expliciete datatypes door het type var en voer het programma uit.

### 5.2.5 Final variables of constanten

Final variables zijn variabelen waarvan men de inhoud niet meer kan wijzigen zodra die is toegekend. Ze hebben dezelfde functie als constanten in andere programmeertalen.

Een *final variable* wordt als volgt gedeclareerd:

```
final int CONSTANT;
```

De initialisatie kan gebeuren tijdens de declaratie of erna.

```
final int CONSTANT = 7;
```

of

```
final int CONSTANT;  
CONSTANT = 7;
```

Als men nadien de waarde toch tracht te veranderen, krijgt men een foutmelding van de compiler.

Constanten worden gewoonlijk in hoofdletters weergegeven. Indien de naam uit meerdere woorden bestaat, worden deze gescheiden door een *underscore* (\_).

MY\_CONSTANT

#### Opdracht 2: Final variables

In deze opdracht gebruiken we een *final variable*.

- Voeg in vorige oefening een constante toe:

```
final double PI = 3.14;
```

- Tracht de waarde te veranderen door de volgende regel toe te voegen:

```
final double PI = 3.14;  
PI = 3.15;
```

- Compileer en zie welke foutmelding de compiler geeft.

### 5.2.6 Typeconversie

Gegevens van een bepaald type kunnen geforceerd worden naar een ander type. Dit noemt men *type casting* of typeconversie.

Typeconversie wordt door de compiler meestal automatisch toegepast, zolang er geen gegevens verloren kunnen gaan.

Je kan het vergelijken met het overgieten van de inhoud van één doosje in een ander doosje. Als het tweede doosje groter is dan het eerste, kan het overgieten probleemloos verlopen. Als het tweede doosje echter kleiner is, lopen we het risico dat het overloopt en dat er dus gegevens verloren gaan.

Indien er gevaar is voor verlies van gegevens zal de compiler weigeren dit te doen.

Neem het volgende voorbeeld:

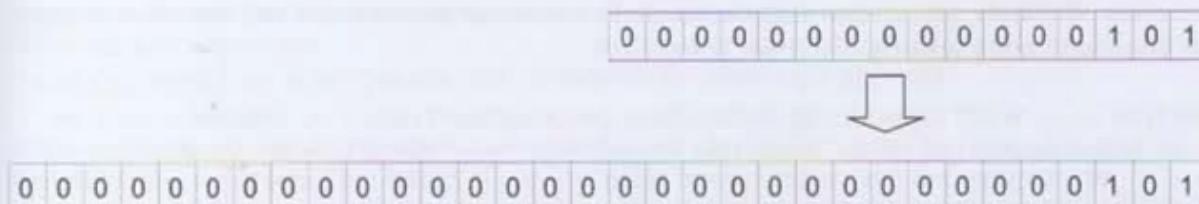
```
short aShort = 5;
int anInt;

anInt = aShort;
```

De waarde van `aShort` wordt automatisch geconverteerd van `short` naar `int`.

Deze conversie verloopt probleemloos omdat er geen gegevens verloren kunnen gaan bij de conversie. Iedere `short`-waarde past namelijk in een `int`-waarde.

We zien dit als we naar de bit-voorstelling van de getallen kijken:



Afbeelding 34: Conversie van `short` naar `int`

Dit is anders in het volgende voorbeeld:

```
short aShort;
int anInt = 600_000;

aShort = anInt;
```

Vermits een `int` waarden kan bevatten die niet passen in een `short`, zal de conversie niet automatisch gebeuren. De compiler zal een foutmelding geven.

De bit-voorstelling ziet er als volgt uit:



Afbeelding 35: Conversie van `int` naar `short`

We zien hier dat er gegevens verloren gaan.

Toch is het mogelijk deze conversie wel te forceren, met het mogelijke risico van gegevensverlies. Het is dan de verantwoordelijkheid van de programmeur om hier rekening mee te houden.



De syntax van een expliciete typeconversie is de volgende:

```
(type) value
```

De waarde van een bepaald type wordt omgezet naar een ander type door het nieuwe type tussen rondende haakjes voor de waarde te zetten.

Indien het nieuwe datatype minder bits telt, dan worden de overtollige bits gewoon weggegooid. Indien het nieuwe datatype meer bits telt, dan wordt de hoogste bit (tekenbit) gekopieerd naar de nieuwe bits. Op die manier blijft het teken van getallen behouden.

We nemen nogmaals ons voorbeeld:

```
short aShort;  
int anInt = 600_000;  
  
aShort = (short) anInt;
```

Indien de variabele `anInt` een waarde bevat die buiten de grenzen valt van een `short`, dan zal de resulterende waarde niet meer correct zijn.

Het type `char` wordt indien nodig automatisch geconverteerd naar `int`. Hoewel `short` en `char` evenveel bits gebruiken, is het niet mogelijk een `char` toe te kennen aan een `short`:

```
short aShort;  
int anInt;  
char aChar = 'a';  
  
aShort = aChar; // Wrong  
anInt = aChar; // Good
```

De reden hiervoor is dat een `char` beschouwd kan worden als een tekenloos 16-bits-getal. `short` daarentegen is een 16-bits getal met teken. Om de waarde van een `char` correct te behouden moet hij dus omgezet worden in een 32-bits `int`.

**Opmerking:** Men kan geen conversie doen tussen een `boolean` en een ander primitief datatype.

### Opdracht 3: Type-conversie

- Voeg aan vorige opdracht de volgende code toe:

```
aByte = aShort;
```

- Compileer de broncode en zie welke foutmelding de compiler geeft.
- Pas vervolgens een expliciete typeconversie toe.

```
aByte = (byte) aShort;
```

- Voer het programma uit en ga na welke waarde voor `aByte` wordt afdrukkt.
- Wijzig de initiële waarde van `aShort` in 115 en voer het programma opnieuw uit.
- Ken `aChar` toe aan `aShort` en ga na welke foutmelding de compiler geeft.
- Ken `aChar` toe aan `anInt`.

### 5.3 Operatoren

Zodra we in het geheugen plaats gereserveerd hebben voor onze gegevens, kunnen we starten met het bewerken van deze gegevens. De basisbewerkingen worden uitgevoerd door middel van operatoren. Een operator neemt een of meerdere waarden en voert er een bewerking op uit. Dit resulteert in een nieuw gegeven waar we dan vervolgens weer iets mee kunnen gaan doen: bijvoorbeeld gebruiken in een volgende bewerking of opslaan in een andere variabele.

We geven nu al even een voorbeeld om het anschouwelijk te maken:

```
int sum = a + b;
```

We tellen de inhoud van variabele `a` en variabele `b` op en dit tussenresultaat steken we in de variabele `sum`.

Operatoren zijn dus bewerkingstekens die gebruikt worden om bewerkingen te doen tussen gegevens die we hier de operanden noemen. Een operand kan zowel een variabele, een *literal* als een object zijn.

Voorlopig maken we enkel gebruik van variabelen en *literals* als operand.

Een bewerking op een of meerdere operanden geeft steeds een resultaat terug. De waarde en het type van het resultaat hangt af van het soort bewerking en van het type van de operanden.

We kunnen de operatoren onderverdelen volgens het aantal operanden:

1. **Unaire** operatoren: deze voeren een bewerking uit op slechts één operand.
2. **Binaire** operatoren: deze voeren een bewerking uit op twee operanden.
3. **Ternaire** operatoren: deze voeren een bewerking uit op drie operanden.

De algemene syntax ziet er als volgt uit:

Voor de unaire operatoren:

```
operand1 operator  
operator operand1
```

Voor de binaire operatoren:

```
operand1 operator operand2
```

Voor de ternaire operatoren:

```
operand1 operator_deel1 operand2 operator_deel2 operand3
```

We kunnen de operatoren ook onderverdelen volgens het soort bewerking dat ze uitvoeren:

1. **Rekenkundige** operatoren
2. **Relationele** operatoren
3. **Logische** operatoren
4. **Shift**-operatoren
5. **Bit**-operatoren
6. **Toekenning**operatoren
7. Andere operatoren

We zullen deze laatste indeling gebruiken.

### 5.3.1 Rekenkundige operatoren

De rekenkundige operatoren dienen om rekenkundige bewerkingen uit te voeren op numerieke waarden. De operanden zijn in dit geval getallen en het resultaat van de bewerking is tevens een getal.

Onderstaande tabel bevat een lijst van de rekenkundige operatoren die in Java gebruikt kunnen worden.

<b>Binaire operatoren</b>		
Operator	Gebruik	Resultaat
+	op1 + op2	De som van op1 en op2.
-	op1 - op2	Het verschil van op2 en op1.
*	op1 * op2	Het product van op1 en op2.
/	op1 / op2	Het quotiënt van op1 en op2.
%	op1 % op2	op1 modulo op2 De rest van de deling van op1 door de absolute waarde van op2.
<b>Unaire operatoren</b>		
++	++op	Verhoogt de waarde van op eerst met 1 en gebruikt daarna de verhoogde waarde van op. Het resultaat is de verhoogde waarde van op.
++	op++	Gebruikt eerst de waarde van op en verhoogt ze daarna met 1. Het onmiddellijke resultaat is op maar nadien wordt de waarde van op wel met 1 verhoogd.
--	--op	Verlaagt de waarde van op eerst met 1 en gebruikt daarna de verlaagde waarde van op. Het resultaat is de verlaagde waarde van op.
--	op--	Gebruikt eerst de waarde van op en verlaagt ze daarna met 1. Het onmiddellijke resultaat is op maar nadien wordt de waarde van op wel met 1 verlaagd.
-	-op	De negatieve waarde van op.
+	+op	Promoveert een byte, short of char naar int.

Tabel 5: Rekenkundige operatoren

De operatoren ++ en -- kunnen voor of achter de *operand* geplaatst worden. Dit geeft een verschillend gedrag. We illustreren dit met een voorbeeld:

```
int a = 5;
int b = ++a;
```

In dit geval zal eerst de waarde van a verhoogd worden tot 6 en nadien wordt deze waarde toegekend aan b. Na afloop zijn zowel a als b gelijk aan 6.

Nemen we daarentegen het volgende voorbeeld:

```
int a = 5;
int b = a++;
```

In dit geval zal eerst de waarde van `a` (5) gebruikt worden om toegekend te worden aan de variabele `b`. Nadien wordt `a` verhoogd naar 6. Na afloop is `a` gelijk aan 6 en `b` gelijk aan 5.

**Opgelet:** De operatoren + - \* / % hebben geen invloed op de waarde van de operanden zelf. De operatoren ++ en -- daarentegen veranderen de waarde van de operanden wel. Deze kunnen daarom ook enkel toegepast worden op variabelen, niet op letterlijke waarden.

Alle rekenkundige operatoren geven telkens een numerieke waarde als onmiddellijk resultaat. Het datatype van het resultaat is minimaal int tenzij een van de operanden een type heeft dat hoger in rang is. In dit geval is het uiteindelijke type datgene van de operand met de hoogste rang.

Rangorde	Type
1	double
2	float
3	long
4	<b>int</b>
5	short
6	byte

minimale type

Tabel 6: De rangorde van datatypes

Bijvoorbeeld:

```
byte + byte = int
byte + float = float
short * byte = int
float * float = float
```

De enige **uitzondering** hierop vormen de operatoren ++ en --. Hierbij is het datatype van het resultaat gelijk aan het datatype van de operand.

De volgende code geeft een foutmelding:

```
byte a = 5;
byte b = -a;
```

De - operator geeft als resultaat een int en die kan men niet toekennen aan een byte.

De volgende code is wel correct:

```
byte a = 5;
byte b = --a;
```

De -- operator geeft een resultaat van hetzelfde type als de operand, in dit geval een byte.

**Opmerking:** Het kan gebeuren dat het resultaat van een bewerking te groot is voor het

resulterende datatype. In dat geval is het resultaat niet correct.

We nemen het volgende voorbeeld:

```
public class DataTypes {
    public static void main(String[] args) {
        int number1 = 2147483645;
        int number2 = 2147483642;
        int result;

        result = number1 * number2;

        System.out.println(number1);
        System.out.println(number2);
        System.out.println(result);
    }
}
```

Het datatype van het resultaat is `int`. Het resultaat zelf is echter een getal dat te groot is voor een `int`. De compiler geeft geen foutmelding en de uitvoering van het programma geeft als resultaat `-2147483630`, hetgeen foutief is.

Dit probleem kan als volgt opgelost worden:

```
...
int number1 = 2147483645;
int number2 = 2147483642;
long result;
result = (long)number1 * number2;
...
```

De variabele `result` maken we van het datatype `long`, zodat deze groot genoeg is om het resultaat van de vermenigvuldiging te bevatten. Dit volstaat echter niet omdat het resultaat van de vermenigvuldiging van twee integers ook een `integer` is. Het loopt dus al fout voordat we het resultaat toekennen aan de variabele `result`. Daarom converteren we een van de integers eerst naar een `long`. Het resultaat van de vermenigvuldiging van een `long` met een `int` is namelijk een `long`.

We krijgen het volgende correcte resultaat `4611685999100035090`.

We nemen nog een ander voorbeeld:

```
public class DataTypes2 {
    public static void main(String[] args) {
        byte number1 = 7;
        byte number2 = 5;
        byte result;

        result = number1 * number2;

        System.out.println(number1);
        System.out.println(number2);
        System.out.println(result);
    }
}
```

Dit geeft ook een foutmelding bij de compilatie.

```
types2.java:9: possible loss of precision
found   : int
required: byte
        result = number1 * number2;
                           ^
1 error
```

De vermenigvuldiging van twee bytes geeft dus minimaal een integer. Deze integer kunnen we niet zomaar toekennen aan een byte.

Dit probleem stelt zich niet als we *literals* gebruiken.

```
byte result = 5 * 7;
```

Hoewel *literals* als integers beschouwd worden, ziet de compiler dat er in dit geval geen probleem is en zal hij de nodige conversie doen.

Zodra het resultaat van de bewerking niet meer geconverteerd kan worden zonder gegevensverlies, zal de compiler een foutmelding geven:

```
byte result = 5 * 70; // Wrong
```

Indien karakters gebruikt worden met rekenkundige operatoren, worden deze omgezet in de overeenkomstige integer-waarde. De enige uitzondering hierop zijn de operatoren `++` en `--`.

```
char c = 'a';
c = c + 1; // Wrong: result is an int
c++;        // Good: result is a char
```

#### **Opdracht 4: Rekenkundige operatoren gebruiken**

In deze opdracht maken we gebruik van de rekenkundige operatoren. We kunnen het resultaat van een uitdrukking op het scherm brengen met de methode `System.out.println()`. Tussen de ronde haken plaatsen we de uitdrukking waarvan we het resultaat willen zien.

Bijvoorbeeld:

```
System.out.println(3 + 8);
```

We kunnen dit resultaat laten voorafgaan door tekst. In dit geval plaatsen we een `+` teken tussen de string die de tekst bevat en de uitdrukking, waarbij we de gehele uitdrukking eerst tussen ronde haken zetten.

Bijvoorbeeld:

```
System.out.println("The sum of 3 + 8 " + (3 + 8));
```

Om gegevens van het toetsenbord te kunnen lezen, gebruiken we de volgende code:

```
package exercise05_04;
import java.util.*;

public class Arithmetic {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        int intValue = keyboard.nextInt();

        ...

        keyboard.close();
    }
}
```

We plaatsen de regel `import java.util.*;` vlak onder de definitie van het pakket. Het inlezen gebeurt met `keyboard.nextInt()`. Er zijn ook nog de volgende varianten voor de andere datatypes:

```
keyboard.nextByte()
keyboard.nextShort()
keyboard.nextLong()
keyboard.nextFloat()
keyboard.nextDouble()
```

De juiste uitleg betreffende deze regels code volgt later; op dit moment maken ze de opdrachten enkel wat interactiever.

- Maak een programma met twee variabelen `number1` en `number2` van het type `int`. Initialiseer deze variabelen met een bepaalde waarde. Druk vervolgens het volgende op het scherm af:
  - de som van beide getallen
  - het verschil van beide getallen
  - het product van beide getallen
  - de deling van beide getallen
  - de rest van de deling van beide getallen
  - `--number1` , `number1--` , `++number1` , `number1++`
- Test het gebruik van de operatoren `++` en `--` door ze afwisselend voor en achter de operand te zetten en het resultaat op het scherm te brengen.
- Maak een variabele van het type `char`. Tel een waarde bij het karakter op en druk het af op het scherm.

#### **Opdracht 5: Het datatype van tussenresultaten**

In deze opdracht gaan we na wat het datatype van het resultaat van een bewerking is.

- Maak een programma met drie bytes. Ken aan de derde byte het product van de eerste twee bytes toe. Compileer en zie welke foutmelding de compiler geeft.
- Vermenigvuldig twee integers met de volgende waarden: 2147483645 en 2147483642. Ken het resultaat aan een derde integer toe en zie wat het resultaat is.
- Los het probleem op door het gebruik van de juiste datatypes en *casting*.

#### **5.3.2 Relationale operatoren**

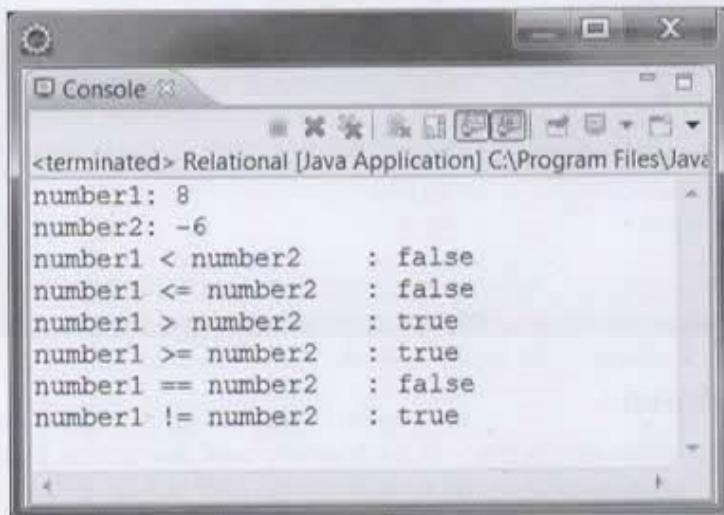
Relationale operatoren bepalen de relatie tussen twee operanden en geven als resultaat de waarde `true` of `false` (boolean).

Binaire operatoren		
Operator	Gebruik	Resultaat
>	op1 > op2	true als op1 groter is dan op2
>=	op1 >= op2	true als op1 groter of gelijk is aan op2
<	op1 < op2	true als op1 kleiner is dan op2
<=	op1 <= op2	true als op1 kleiner of gelijk is aan op2
==	op1 == op2	true als op1 gelijk is aan op2
!=	op1 != op2	true als op1 verschillend is van op2

Tabel 7: Relationale operatoren

**Opdracht 6: Relationale operatoren gebruiken**

- Maak een programma met twee variabelen. Initialiseer deze variabelen met een bepaalde waarde. Ga vervolgens met alle relationele operatoren na welke de relatie is tussen de twee getallen. Breng het resultaat telkens op het scherm.



```

<terminated> Relational [Java Application] C:\Program Files\Java
number1: 8
number2: -6
number1 < number2      : false
number1 <= number2     : false
number1 > number2      : true
number1 >= number2     : true
number1 == number2      : false
number1 != number2      : true
  
```

**5.3.3 Logische operatoren**

Logische operatoren worden gebruikt om logische combinaties te maken van operanden van het type boolean. Het resultaat is steeds een booleaanse waarde (*true* of *false*).

Binaire operatoren		
Operator	Gebruik	Resultaat
&&	op1 && op2	true als op1 EN op2 true zijn
	op1    op2	true als op1 OF op2 true is
Unaire operatoren		
!	!op	true als op false is (inverse)

Tabel 8: Logische operatoren

**Opgelet:** Indien bij de evaluatie van de eerste operand het resultaat reeds bepaald is, dan wordt de tweede operand niet meer geëvalueerd. Dit kan van belang zijn wanneer de tweede

operand uit een bewerking bestaat. Die bewerking wordt dan niet meer uitgevoerd. Men noemt dit *short circuit evaluation*.

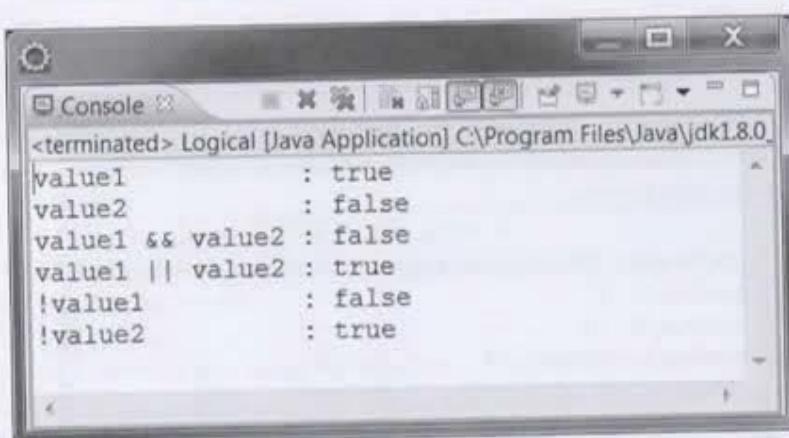
Voorbeeld:

```
int number1 = 3;
int number2 = 30;
System.out.println((number1 < 5) || (++number2 > 10));
System.out.println(number2);
```

Indien de eerste voorwaarde true is, dan wordt de tweede niet meer geëvalueerd en wordt dus ook de bewerking `++number2` niet meer uitgevoerd.

### Opdracht 7: Logische operatoren gebruiken

- Maak een programma met twee boolean variabelen met een initiële waarde. Gebruik de logische operatoren en breng het resultaat op het scherm.



```
<terminated> Logical [Java Application] C:\Program Files\Java\jdk1.8.0
value1      : true
value2      : false
value1 && value2 : false
value1 || value2 : true
!value1      : false
!value2      : true
```

#### 5.3.4 Shift-operatoren

De shift-operatoren verschuiven de bits van de operand over een aantal posities. Beide operanden zijn gehele getallen en het resultaat is tevens een geheel getal.

Binaire operatoren		
Operator	Gebruik	Resultaat
>>	op1 >> op2	De waarde van op1 binair naar rechts verschoven over het aantal posities aangegeven door op2. De inschuivende bit is gelijk aan de hoogste bit. Met andere woorden het teken van het getal wordt behouden.
<<	op1 << op2	De waarde van op1 binair naar links verschoven over het aantal posities aangegeven door op2. Er wordt rechts een 0 ingeschoven.
>>>	op1 >>> op2	De waarde van op1 binair verschoven naar rechts over een aantal posities aangegeven door op2. De inschuivende bit is gelijk aan 0. Met andere woorden de tekenbit wordt niet behouden.

Tabel 9: Binaire operatoren

**Opgelet:** De shift-operatoren hebben geen invloed op de waarde van de operand zelf. Ze

bepalen enkel het resultaat van de bewerking.

De *shift*-operatie wordt intern steeds uitgevoerd op basis van een *int* of een *long*. Dit heeft tot gevolg dat een *byte* en een *short* eerst geconverteerd worden naar een *int*. Bij deze conversie wordt de hoogst beduidende bit gekopieerd naar de toegevoegde bits. In het geval van negatieve getallen kan dit onverwachte resultaten geven. Daarom is het aan te raden de *shift*-bewerkingen steeds op een *int* of een *long* te doen.

Het resultaat is minimaal een *int*.

Indien de tweede operand groter is dan het aantal bits van de eerste operand, wordt eerst een modulobewerking uitgevoerd met het aantal bits.

$$1234 \gg 34 = 1234 \gg (34 \% 32) = 1234 \gg 2 = 308$$

In feite worden steeds de 5 (*int*) of 6 (*long*) minst beduidende bits genomen van de tweede operand om te bepalen hoeveel posities er geschoven moet worden. In het geval van een negatieve waarde van de tweede operand kan dit onverwachte resultaten opleveren:

$$1 \ll -1 = 1 \ll 31 = -2147483648$$

De operator *>>>* is de *unsigned shift*. Deze bestaat niet in C/C++. In C/C++ kent men namelijk ook getallen zonder teken (*unsigned int*). De *shift*-bewerking verschilt daarbij naargelang het om een getal met of zonder teken gaat.

Java kent alleen getallen met teken. Om toch dezelfde *shift*-bewerking te kunnen hebben, zoals bij getallen zonder teken in C/C++, heeft men de *unsigned shift* operator toegevoegd.

Voorbeeld:

Het getal 5 wordt binair als volgt voorgesteld:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Als we hier de bewerking  $\gg 1$  op uitvoeren, krijgen we:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Alle bits worden één positie naar rechts geschoven. De hoogst beduidende bit wordt gekopieerd en de laagst beduidende bit gaat verloren.

Het resultaat is het getal 2.

Als we de bewerking  $\ll 1$  op het getal 5 uitvoeren krijgen we het volgende:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

Dit is het getal 10.

Alle bits worden één positie naar links geschoven. De hoogst beduidende bit gaat verloren en de laagst beduidende bit krijgt de waarde 0.

Het getal  $-5$  wordt binair als volgt voorgesteld:



Men gebruikt hiervoor het *two's complement*. Hierbij wordt een positieve waarde geïnverteerd en verhoogd met 1.

$-5 >> 1$  wordt dan:



wat overeenkomt met de waarde  $-3$ .

De meest beduidende bit wordt hierbij gekopieerd.

$-5 >>> 1$  daarentegen wordt:



wat overeenkomt met de waarde  $2147483645$ .

Bij deze bewerking wordt steeds een 0 binnengeschoven.

$-5 << 1$  wordt:



Dit is gelijk aan de waarde  $-10$ .

### Het praktisch gebruik van de *shift*-operatoren.

*Shift*-operatoren worden onder andere gebruikt om op een snelle manier getallen te vermenigvuldigen of te delen met een macht van twee. Stel dat we het getal 5 met 2 willen vermenigvuldigen, dan kunnen we dat op de volgende manier doen:

$5 * 2$

Deze bewerking is echter rekenintensief en kan veel sneller en efficiënter gebeuren:

$5 << 1$

De shiftbewerking is doorgaans veel sneller dan de rekenkundige bewerking.

Indien we een getal willen delen door 1024 kan dat als volgt:

```
number/1024
```

Een efficiëntere manier is:

```
number>>10
```

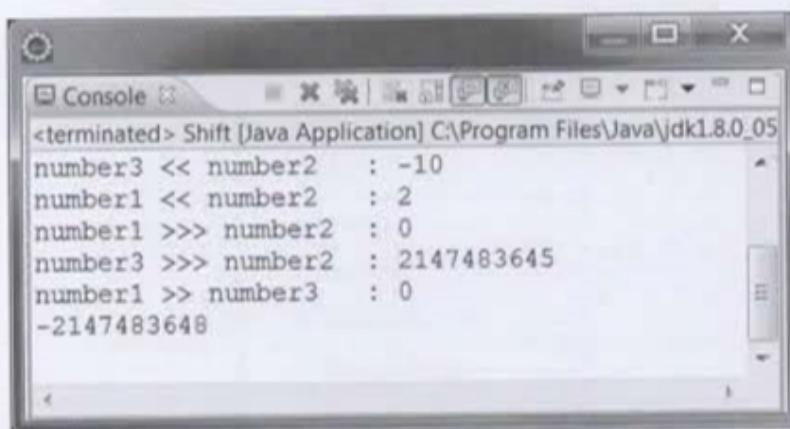
1024 is namelijk gelijk aan  $2^{10}$ .

Een snelle vermenigvuldiging met 10 ziet er met een shiftbewerking als volgt uit:

```
(( (number<<2)+number)<<1)
```

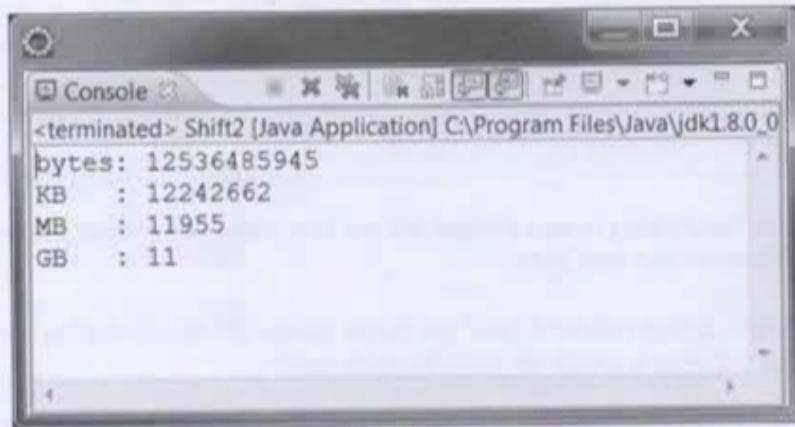
#### **Opdracht 8: Shift-operatoren gebruiken**

- Maak een programma met drie getallen. Initialiseer de getallen met een bepaalde waarde en pas de shift-operatoren toe.



```
Console <terminated> Shift [Java Application] C:\Program Files\Java\jdk1.8.0_05
number3 << number2 : -10
number1 << number2 : 2
number1 >>> number2 : 0
number3 >>> number2 : 2147483645
number1 >> number3 : 0
-2147483648
```

- Pas de operatoren `>>>` en `>>` toe op een negatief getal en evalueer het resultaat.
- Maak een getal dat de grootte van een harde schijf opslaat in een variabele (`long`). Druk de grootte af in aantal bytes, aantal kilobytes, aantal megabytes en aantal gigabytes. Maak telkens gebruik van de shift-operator.



```
Console <terminated> Shift2 [Java Application] C:\Program Files\Java\jdk1.8.0_05
bytes: 12536485945
KB : 12242662
MB : 11955
GB : 11
```

#### **5.3.5 Bit-operatoren**

Bit-operatoren voeren bitgewijze bewerkingen uit op de operanden. De operanden moeten steeds gehele getallen zijn. Intern gebeurt de bewerking op basis van een `int` of een `long`. Een `byte` of een `short` zal dus eerst geconverteerd worden naar een `int`.

<b>Binaire operatoren</b>		
<b>Operator</b>	<b>Gebruik</b>	<b>Resultaat</b>
&	op1 & op2	Bitgewijze EN tussen op1 en op2
	op1   op2	Bitgewijze OF tussen op1 en op2
^	op1 ^ op2	Bitgewijze EXCLUSIEVE OF tussen op1 en op2
<b>Unaire operatoren</b>		
~	~op	Bitgewijze complement van op

Tabel 10: Bit-operatoren

Voorbeelden:

<b>Decimaal</b>	<b>Binair</b>
5	0 0 0 0 0 1 0 1
85	0 1 0 1 0 1 0 1
5 & 85 = 5	0 0 0 0 0 1 0 1
5   85 = 85	0 1 0 1 0 1 0 1
5 ^ 85 = 80	0 1 0 1 0 0 0 0
~5 = -6	1 1 1 1 1 0 1 0
~85 = -86	1 0 1 0 1 0 1 0

Tabel 11: Voorbeeld van bit-operatoren

Zowel voor de binaire als de unaire operatoren geldt dat het resultaat van de bewerking minimaal een integer is. Als een van beide operanden van een hogere rangorde is, zal het resultaat het type van de hoogste rang aannemen.

De volgende code geeft daarom een foutmelding:

```
byte number1 = 5;
byte number2;
number2 = ~number1;           // Wrong
number2 = (byte) ~number1;   // Good
```

Het resultaat van de bewerking is een integer en die kan men niet zonder expliciete typeconversie toekennen aan een byte.

De operanden worden geconverteerd naar het juiste type voor de uitvoering van de bewerking. Bij deze conversie wordt de tekenbit behouden.

De bitoperatoren & | en ^ kunnen ook gebruikt worden met booleans. In dit geval worden de uitdrukkingen die resulteren in een boolean steeds geëvalueerd.

We nemen nogmaals het voorbeeld van de logische operatoren:

```
(number1 < 5) | (++number2 > 10)
```

In dit geval wordt `number2` wel verhoogd ook al is `number1` kleiner dan 5.

### Het praktisch gebruik van de bit-operatoren.

Bit-operatoren worden enkel gebruikt indien men operaties wil uitvoeren op individuele bits in een variabele.

We nemen een concreet voorbeeld. Stel dat we een Java-applicatie moeten schrijven voor een of ander domoticaproject. De centrale computer regelt onder andere de verlichting in een huis. Het is dan mogelijk de algemene verlichtingstoestand van de woning samen te brengen in één variabele:

```
int light;
```

De variabele `light` bestaat uit 32 bits. Iedere bit geeft aan of het licht in een bepaalde ruimte aan (1) of uit is (0). Voor dit voorbeeld beperken we ons tot de acht minst beduidende bits.

Bit	Ruimte
0	Hal
1	Living
2	Keuken
3	Veranda
4	Slaapkamer
5	Wasplaats
6	Badkamer
7	Kelder

Tabel 12: Bit-toekenning per ruimte

Indien we het licht in de keuken willen aandoen, moeten we de derde bit op 1 zetten. Hiervoor gebruiken we een OR-operatie.

```
light = light | 0x4;
```

<code>light</code>	<code>0 1 1 0 0 0 0 1</code>
<code> </code>	
<code>0x04</code>	<code>0 0 0 0 0 1 0 0</code>
<code>=</code>	
<code>light</code>	<code>0 1 1 0 0 1 0 1</code>

Willen we het licht terug uitdoen, dan moeten we de derde bit terug op 0 zetten. Hiervoor gebruiken we een AND-operatie.

```
light = light & 0xFB;
```

<code>light</code>	<code>0 1 1 0 0 1 0 1</code>
<code>&amp;</code>	
<code>0xFB</code>	<code>1 1 1 1 1 0 1 1</code>
<code>=</code>	

light	0	1	1	0	0	0	1
-------	---	---	---	---	---	---	---

Merk op dat 0xFB het omgekeerde is van 0x4. We kunnen dus ook schrijven:

```
light = light & ~0x4;
```

We kunnen verder aan elke ruimte een bepaald bitmasker koppelen. De keuken is de derde bit en het bitmasker voor de keuken is daarom 0x4. Dit bitmasker kennen we gemakkelijkheidshalve aan een constante toe.

Bit	Ruimte	Constante	Bitmasker
0	Hal	HALL	0x01 00000001
1	Living	LIVINGROOM	0x02 00000010
2	Keuken	KITCHEN	0x04 00000100
3	Veranda	VERANDA	0x08 00001000
4	Slaapkamer	BEDROOM	0x10 00010000
5	Wasplaats	LAUNDRY	0x20 00100000
6	Badkamer	BATHROOM	0x40 01000000
7	Kelder	CELLAR	0x80 10000000

Tabel 13: Bitmaskers per ruimte

De Java-code:

```
final int HALL      = 0x01;
final int LIVINGROOM = 0x02;
final int KITCHEN    = 0x04;
final int VERANDA    = 0x08;
final int BEDROOM    = 0x10;
final int LAUNDRY    = 0x20;
final int BATHROOM   = 0x40;
final int CELLAR     = 0x80;
```

Het licht in de badkamer aandoen, kan dan als volgt:

```
light = light | BATHROOM;
```

Het licht in de badkamer uitdoen:

```
light = light & ~BATHROOM;
```

Indien we willen controleren of het licht in de slaapkamer aan is, drukken we het volgende op het scherm af:

System.out.println(( light & BEDROOM) == BEDROOM)
---

De resulterende waarde is namelijk verschillend van 0 als de bit voor het licht van de slaapkamer op 1 staat.

Stel dat we het licht op de benedenverdieping in één keer willen uitdoen. We maken dan

eerst een bitmasker voor de gehele benedenverdieping die bestaat uit de hal, de keuken, de living en de veranda. Door middel van een OF-bewerking maken we een gecombineerd bitmasker,

```
final int DOWNSTAIRS = HALL | KITCHEN | LIVINGROOM | VERANDA;
```

Dit resulteert in de waarde 0x0F of 00001111.

Het licht op de benedenverdieping doen we dan als volgt uit:

```
light = light & ~DOWNSTAIRS;
```

Voor diefstalpreventie laat men bepaalde lichten op regelmatige tijdstippen uit- en aangaan. We maken daarom een bitmasker voor deze lichten.

```
final int PREVENTION = HALL | LIVINGROOM | BEDROOM;
```

Als initiële toestand doen we het licht in de hal en de slaapkamer aan:

```
light = HALL | BEDROOM;
```

We willen nu dat om de 20 minuten de toestand van deze lichten wisselt. Hiervoor gebruiken we de EXCLUSIEVE OF-bewerking.

```
light = light ^ PREVENTION;
```

Deze bewerking kan om de 20 minuten uitgevoerd worden, ongeacht de toestand van de verlichting op dat moment.

Samengevat:

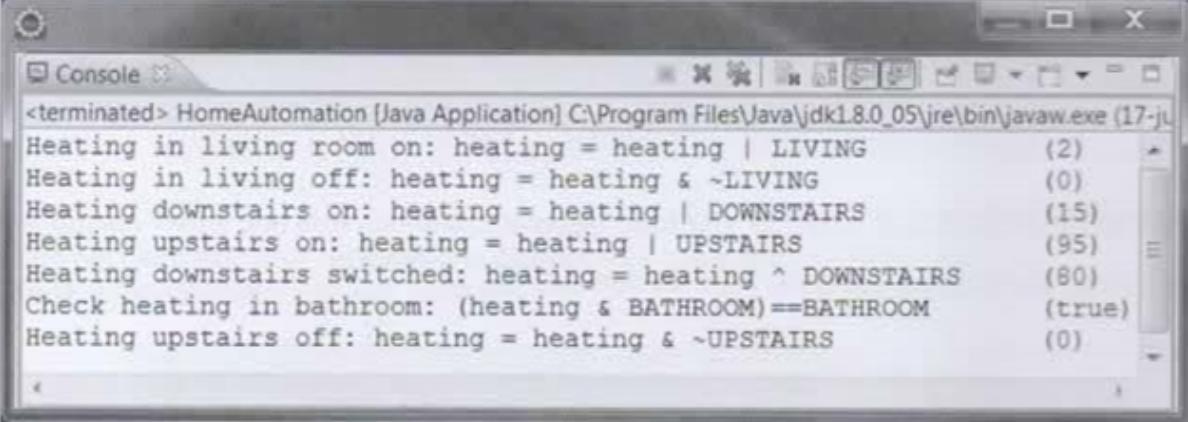
Actie	Operator
bit(s) op 1 zetten	OF-bewerking met het bitmasker van de bit(s)
bit(s) op 0 zetten	EN-bewerking met het geïnverteerde bitmasker van de bit(s)
bit(s) wisselen	EXCLUSIEVE OF-bewerking met het bitmasker van de bit(s)
bit(s) controleren	EN-bewerking met het bitmasker van de bit(s)

Tabel 14: Samenvatting van de bit-operatoren

### Opdracht 9: Bit-operatoren gebruiken

In deze opdracht gaan we de domoticatoepassing uitbreiden met een variabele die de verwarming regelt. In elk vertrek kan men de verwarming afzonderlijk op- en afzetten.

- Maak een variabele `heating` en de nodige bitmaskers voor de afzonderlijke kamers.
- Maak tevens bitmaskers per verdieping.
- Schakel de verwarming in de living in en druk het resultaat op het scherm af.
- Schakel de verwarming terug uit en druk het resultaat op het scherm af.
- Schakel de verwarming beneden aan.
- Schakel de verwarming boven aan.
- Verwissel de toestand beneden.
- Controleer de verwarming in de badkamer.
- Schakel de verwarming boven uit.



```

Console > Heating in living room on: heating = heating | LIVING          (2)
Console > Heating in living off: heating = heating & ~LIVING        (0)
Console > Heating downstairs on: heating = heating | DOWNSTAIRS      (15)
Console > Heating upstairs on: heating = heating | UPSTAIRS         (95)
Console > Heating downstairs switched: heating = heating ^ DOWNSTAIRS (80)
Console > Check heating in bathroom: (heating & BATHROOM)==BATHROOM   (true)
Console > Heating upstairs off: heating = heating & ~UPSTAIRS       (0)
  
```

### 5.3.6 Toekenningsoperatoren

Er is slechts één toekenningsoperator = die de waarde van de tweede operand aan de eerste toekent.

Binaire operatoren		
Operator	Gebruik	Resultaat
=	op1 = op2	op1 krijgt de waarde van op2 toegekend. Het resultaat van de bewerking is de nieuwe waarde van op1.

Tabel 15: De toekenningsoperator

De eerste operand moet een variabele zijn en kan geen *literal* zijn. Indien het datatype van de tweede operand van een hogere rangorde is, zal de compiler een foutmelding geven. Een expliciete typeconversie is dan vereist.

Deze toekenningsoperator wordt vaak gecombineerd met andere operatoren.

Operator	Gebruik	Gelijk aan
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
=	op1  = op2	op1 = op1   op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabel 16: De gecombineerde toekenningsoperator

We nemen nogmaals het voorbeeld van de domoticatoepassing. Het licht in de badkamer deden we als volgt aan:

```
light = light | BATHROOM;
```

Dit kan korter geschreven worden op de volgende manier:

```
light |= BATHROOM;
```

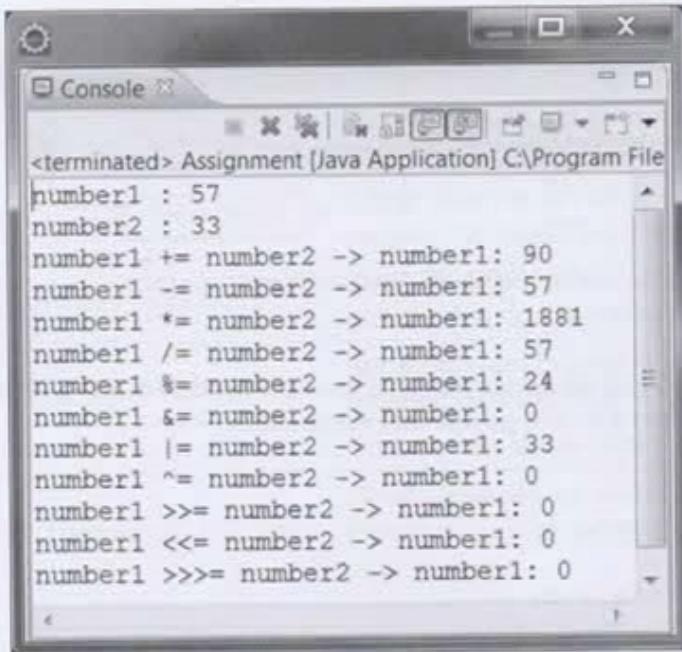
**Opgelet:** bij het gebruik van de gecombineerde toekenningsoperator geldt de regel van het minimale datatype van het tussenresultaat niet. We illustreren dit met een voorbeeld:

```
byte b1 = 5;
byte b2 = 7;

b1 = b1 + b2; // Wrong
b1 += b2; // Good
```

#### **Opdracht 10: De toekenningsoperator gebruiken**

- Vereenvoudig vorige opdracht (domoticaproject) door gebruik te maken van de gecombineerde toekenningsoperator.
- Maak een programma met twee variabelen. Initialiseer de variabelen en pas de gecombineerde toekenningsoperator erop toe.



```
<terminated> Assignment [Java Application] C:\Program File
number1 : 57
number2 : 33
number1 += number2 -> number1: 90
number1 -= number2 -> number1: 57
number1 *= number2 -> number1: 1881
number1 /= number2 -> number1: 57
number1 %= number2 -> number1: 24
number1 &= number2 -> number1: 0
number1 |= number2 -> number1: 33
number1 ^= number2 -> number1: 0
number1 >= number2 -> number1: 0
number1 <= number2 -> number1: 0
number1 >>= number2 -> number1: 0
```

**Opmerking:** Hou er rekening mee dat de waarde van `number1` door de toekenning voortdurend verandert in de loop van het programma.

#### **5.3.7 Conditionele operatoren**

De enige ternaire operator is de conditionele operator `? :`. Dit is eigenlijk een verkorte `if else`. Er wordt hier dus een keuze gemaakt.

Ternaire operatoren		
Operator	Gebruik	Resultaat
?:	op1?op2:op3	Als op1 true is, wordt de waarde van op2 genomen en anders de waarde van op3. op1 moet steeds een boolean zijn.

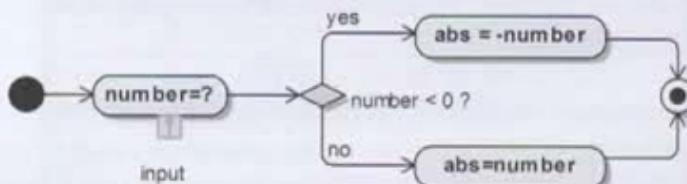
Tabel 17: De conditionele operator

op1 moet een booleaanse waarde bevatten. op2 en op3 moeten van hetzelfde type zijn of probleemloos converteerbaar zijn naar eenzelfde type.

Voorbeeld:

```
int number;
...
int abs = (number<0)?-number:number;
```

Schematisch ziet er dit als volgt uit:



Afbeelding 36: De conditionele operator

Eventuele uitdrukkingen op de plaats van op2 en op3 worden enkel geëvalueerd indien deze waarde geselecteerd is. We geven een voorbeeld:

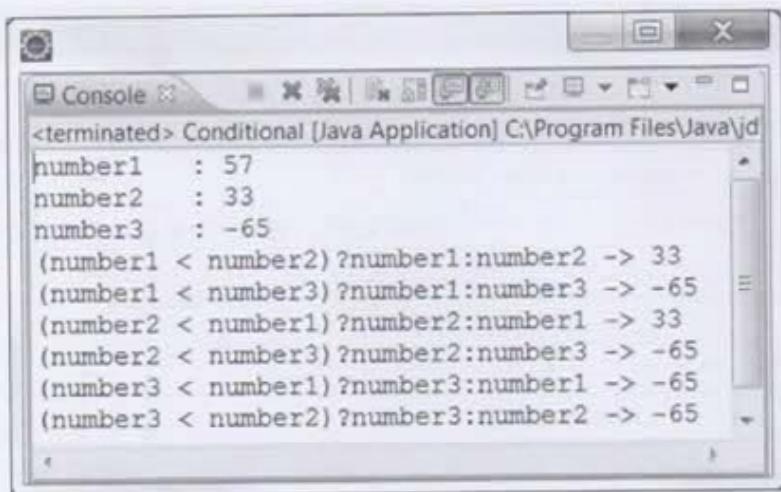
```
int a = 1;
int b = 2;

int value = a<b?++a:++b;
```

Na de uitvoering van de code is a gelijk aan 2 (verhoogd dus) en blijft b gelijk aan 2. We willen hierbij wel opmerken dat dergelijke code best vermeden kan worden.

#### Opdracht 11: De conditionele operator gebruiken

- Maak een programma met een aantal variabelen. Neem de variabelen per twee en druk telkens het kleinste getal af.



```

Console <terminated> Conditional [Java Application] C:\Program Files\Java\jd
number1 : 57
number2 : 33
number3 : -65
(number1 < number2)?number1:number2 -> 33
(number1 < number3)?number1:number3 -> -65
(number2 < number1)?number2:number1 -> 33
(number2 < number3)?number2:number3 -> -65
(number3 < number1)?number3:number1 -> -65
(number3 < number2)?number3:number2 -> -65
    
```

### 5.3.8 Overige operatoren

Er zijn nog enkele andere operatoren die verder in het verloop van de cursus besproken zullen worden.

Operator	Gebruik	Resultaat
.	Object.property	Wordt gebruikt om gekwalificeerde namen te creëren ( <i>qualified names</i> ).
[]	array[x]	Wordt gebruikt om arrays te maken en toegang te krijgen tot elementen in een array.
(params)	(param1, param2)	Geeft een lijst van parameters weer, gescheiden door komma's.
(type)	(type)value	Verandert een bepaalde waarde in een bepaald type (expliciete typeconversie).
new	new Object()	Creëert een nieuw object of een nieuwe array.
instanceof	op instanceof class	Geeft true als op een object is van de klasse class of van een van diens subklassen.

Tabel 18: Overige operatoren

### 5.3.9 Prioriteitsregels

Het is mogelijk meerdere operatoren te combineren.

We geven een voorbeeld:

```
int result = 1 + 2 * 3 - 4;
```

De vraag rijst dan hoe deze uitdrukking geëvalueerd wordt. Het resultaat hangt namelijk af van de volgorde waarin de operanden geëvalueerd worden en de volgorde waarin de bewerkingen uitgevoerd worden.

Voor de volgorde van de bewerkingen zijn er prioriteits- en associativiteitsregels die worden weergegeven in de onderstaande tabel.

Prioriteit	Operatoren	Associativiteit
1	<code>[] . (params)</code>	→
2	<code>expr++ expr--</code>	←
3	<code>++expr --expr +expr -expr ~ !</code>	←
4	<code>new (type)</code>	←
5	<code>* / %</code>	→
6	<code>+ -</code>	→
7	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>	→
8	<code>&lt; &gt; &lt;= &gt;= instanceof</code>	→
9	<code>== !=</code>	→
10	<code>&amp;</code>	→
11	<code>^</code>	→
12	<code> </code>	→
13	<code>&amp;&amp;</code>	→
14	<code>  </code>	→
15	<code>? :</code>	←
16	<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</code>	←

Tabel 19: Prioriteiten en associativiteit van operatoren

De regel is dat een uitdrukking steeds geëvalueerd wordt van links naar rechts. Maar zodra een operand aan zijn rechterzijde ook nog een operator heeft met een hogere prioriteit, dan zal deze met hogere prioriteit eerst uitgevoerd worden en dat bekomen tussenresultaat zal gebruikt worden in de eerste bewerking.

We geven een voorbeeld:

```
int result = 1;
```

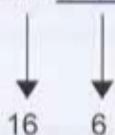
Dit is duidelijk. Er is maar één operator.

We breiden het nu wat uit:

```
int result = 1 + 2;
```

Nu hebben we twee operatoren. We beginnen van links naar rechts en als eerste hebben we de `=`-operator. Deze heeft twee operands: `result` en `1`. Maar de operand `1` heeft aan zijn rechterzijde ook nog de operator `+` die een hogere prioriteit heeft. Deze heeft dus voorrang op het gebruik van de waarde `1`.

```
int result = 1 + 2;
```



Deze optelling wordt dus eerst uitgerekend en het tussenresultaat wordt vervolgens gebruikt. Zo krijgen we dit:

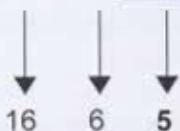
```
int result = 3;
```



16

Laten we het voorbeeld nog wat verder uitbreiden:

```
int result = 1 + 2 * 3;
```

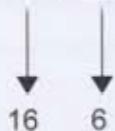


16 6 5

Weer van links naar rechts: = moet onderdoen voor + dus, maar + moet op zijn beurt onderdoen voor \* zodat deze laatste bewerking eerst uitgevoerd zal worden. Dat tussenresultaat zal genomen worden voor de optelling en het tussenresultaat hiervan uiteindelijk voor de toekenning.

We krijgen dan het volgende:

```
int result = 1 + 6;
```



16 6

En vervolgens:

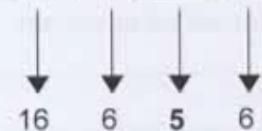
```
int result = 7;
```



16

We breiden het voorbeeld nogmaals wat uit:

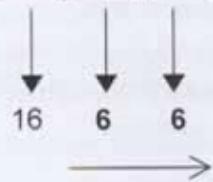
```
int result = 1 + 2 * 3 - 4;
```



16 6 5 6

We gaan weer van links naar rechts en belanden bij de waarde 3. Deze wordt geflankeerd door \* en -. De vermenigvuldiging heeft de hoogste prioriteit en wordt dus eerst uitgerekend:

```
int result = 1 + 6 - 4;
```



16 6 6 →

Nu zitten we met een probleem, want de waarde 6 wordt geflankeerd door twee operatoren met dezelfde prioriteit. In dat geval geldt de associativiteitsregel die zegt dat deze operatoren

van links naar rechts uitgevoerd moeten worden. We krijgen dan dit:

```
int result = 1 + 6 - 4;
           ↓   ↓   ↓
           16  6   6
```

Indien je toch andere prioriteiten wil geven, dien je de bewerkingen tussen ronde haken te plaatsen. Deze worden namelijk altijd eerst geëvalueerd.

Bijvoorbeeld:

```
int result = (1 + 2) * (3 - 4);
```

Hier wordt het resultaat van  $(1 + 2)$  vermenigvuldigd met het resultaat van  $(3 - 4)$ .

Voor de duidelijkheid en leesbaarheid is het aangewezen telkens ronde haken te gebruiken indien de volgorde van de bewerkingen gecompliceerd is, ook al komt dit overeen met de prioriteitsregels en is het strikt genomen niet nodig.

Bijvoorbeeld:

```
int result = 1 + (2 * 3) - 4;
```

### Opdracht 12: Prioriteitsregels

- Bereken zelf de uitkomst van het volgende programma:

```
public class PriorityApp {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int result = ++a * b-- + b < 2 ? --a : ++b;
        System.out.println(result);
    }
}
```

- Controleer je berekening door het programma te maken en uit te voeren.

## 5.4 Uitdrukkingen, statements en blokken

In de programmeertaal Java onderscheiden we uitdrukkingen, *statements* en *blokken*.

### 5.4.1 Uitdrukkingen

Een uitdrukking is een geldige reeks van variabelen, operatoren en functie-aanroepen die resulteren in een enkelvoudige waarde.

In de vorige paragrafen hebben we het gehad over operatoren en operanden. We hebben toen gezegd dat de toepassing van een operator op een of meerdere operanden steeds een bepaalde waarde als resultaat geeft, bijvoorbeeld `number1 + number2`.

Zo'n combinatie van operator en operanden is dus een *uitdrukking*. Het resultaat is een bepaalde waarde.

## 5.4.2 Statements

Een *statement* is een complete programmeerregel afgesloten met een ";".

Men onderscheidt drie soorten *statements*:

### 5.4.2.1 Uitdrukkings-statements

Dit soort *statement* kan op de volgende manieren gevormd worden:

- Een uitdrukking met een **toekenning**:  
vb. number1 = number2 + number3;
- Ieder gebruik van de operatoren -- en ++:  
vb. number1++;
- Het aanroepen van een methode:  
vb. method(number1);
- Creatie van een object:  
vb. button = new Button();

### 5.4.2.2 Declaratie-statements

Een declaratie-*statement* betreft de declaratie van een variabele:

vb. int number = 5;

### 5.4.2.3 Programmaverloop-statement

Deze *statements* betreffen het verloop van het programma:

vb. if else;

Op dit soort *statements* komen we later nog terug.

## 5.4.3 Codeblok

Een codeblok is een groepering van een aantal *statements* die tussen accolades wordt geplaatst.

Voorbeeld

```
{  
    int number = 5;  
    number++;  
    print(number);  
}
```

Dit blok kan op dezelfde manier behandeld worden als één *statement*. Codeblokken worden onder andere gebruikt bij de definitie van methoden en bij *statements* die het programmaverloop bepalen.

Codeblokken hebben invloed op het bereik (scope) van een variabele.

Het bereik van een variabele is het gebied binnen het programma waar men gebruik kan maken van de variabele. Buiten dit gebied is de variabele onbereikbaar. Zo'n gebied wordt bepaald door het codeblok waarbinnen de variabele gedefinieerd wordt.

Men kan het bereik van een variabele op een handige manier grafisch bepalen. Dit veronderstelt dat we de broncode op een consequente manier opmaken, nl. door ieder codeblok een aantal karakters te laten inspringen.

```
{  
    // Codeblok 1  
    ...  
    {  
        // Codeblok 2  
        int anInteger;  
        ...  
        {  
            // Codeblok 3  
            ...  
        }  
        ...  
    }  
    ...  
    // Codeblok 4  
    ...  
}
```

Het bereik van een variabele strekt zich dan uit tot het codeblok waarin de variabele gedeclareerd wordt, inclusief ieder codeblok dat daarbinnen gedefinieerd wordt. Het bereik is dus gelijk aan de code op dezelfde hoogte van de declaratie en alle code rechts daarvan.

Een variabele moet uniek zijn binnen zijn bereik. Er mag met andere woorden binnen zijn bereik geen tweede variabele gedefinieerd worden met dezelfde naam. De enige uitzondering hierop vormen de *member*-variabelen van een klasse. We komen hier later nog op terug.

### **Opdracht 13: Het bereik van variabelen**

In deze opdracht gaan we spelen met het bereik van variabelen.

- Maak een programma met een variabele en druk de variabele af op het scherm.
- Zet de declaratie van de variabele binnen een codeblok.
- Compileer het programma en zie welke foutmelding de compiler of de IDE geeft.
- Zet vervolgens het afdrukken van de variabele in een codeblok.
- Declareer binnen het codeblok een variabele met dezelfde naam maar met een andere waarde. Ga na welke waarde wordt afdrukkt.

## **5.5 Programmaverloop-statements**

### **5.5.1 Inleiding**

Tot nu toe hebben we al gebruikgemaakt van variabelen en bewerkingen erop toegepast. Dit zijn zowat de basisinstructies die een computer kan uitvoeren. In het vorige hoofdstuk hebben we uitgelegd dat programmeren ervan bestaat een probleem te verknippen tot een aantal basisinstructies en vervolgens een volgorde te definiëren waarin deze basisinstructies worden uitgevoerd met mogelijke keuzes en herhalingen.

De opeenvolging van instructies is duidelijk. Deze gebeurt gewoon van boven naar onder: *statement na statement*. Voor het maken van keuzes en herhalingen hebben we evenwel iets extra nodig. Dit zijn de programmaverloop-*statements*. Deze *statements* regelen het verloop van het pad. Het gaat hier hoofdzakelijk om keuzes en herhalingen.

In Java kennen we de volgende programmaverloop-*statements*:

1. if else
2. switch
3. while en do while
4. for-lus

In de volgende paragrafen bespreken we deze *statements*.

### 5.5.2 Het if else statement

Laten we beginnen met het maken van een keuze in het uitvoeringspad. Dit wordt gedaan met het *if else statement*. Dit maakt het mogelijk een bepaald *statement* of codeblok voorwaardelijk uit te voeren.

De syntax ziet er als volgt uit:

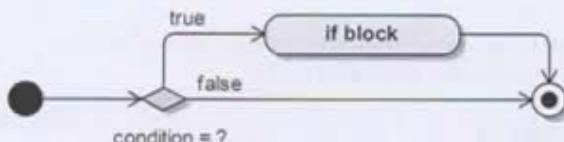
```
if(condition)
    statement;
```

of in het geval van een codeblok:

```
if(condition) {
    statements;
}
```

De voorwaarde is een *boolean* of een uitdrukking met als resultaat een *boolean*. Als de voorwaarde *true* is, wordt het *statement* of het codeblok uitgevoerd.

Schematisch:



Afbeelding 37: Schema van de if

Indien men een verschillend *statement* wil uitvoeren naargelang de uitdrukking *true* of *false* is, kan men een *else* toevoegen.

De basissyntax ziet er als volgt uit:

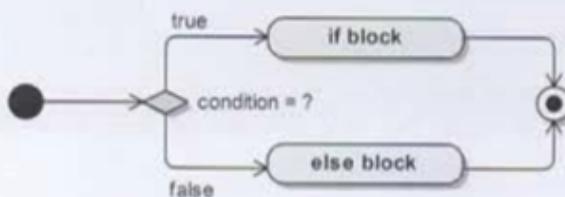
```
if(condition) statement; // Executed if condition is true
else statement; // Executed if condition is false
```

Ook hier kunnen we best gebruikmaken van codeblokken:

```
if(condition) {
    statements; // Executed if condition is true
}
else{
    statements; // Executed if condition is false
}
```

De *statements* in het *else*-codeblok worden uitgevoerd als de voorwaarde *false* is.

Schematisch ziet dit er als volgt uit:



Afbeelding 38: Schema van de if-else

**Opmerking:** Het is een goede programmeertechniek om steeds een codeblok te gebruiken, ook al gaat het slechts om één *statement*. Dit verhoogt de leesbaarheid van het programma en bovendien vermindert het de kans op fouten als later toch een *statement* wordt toegevoegd.

We hernemen hier het voorbeeld dat we reeds hebben aangehaald in het hoofdstuk over programmatielogica:

```
package examples.algorithms.selection;
import java.util.*;

public class Age {
    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);

        System.out.println("Enter your age:");
        var age = keyboard.nextInt();

        if (age >= 18) {
            System.out.println("You are an adult");
        } else {
            System.out.println("You are a child");
        }

        keyboard.close();
    }
}
```

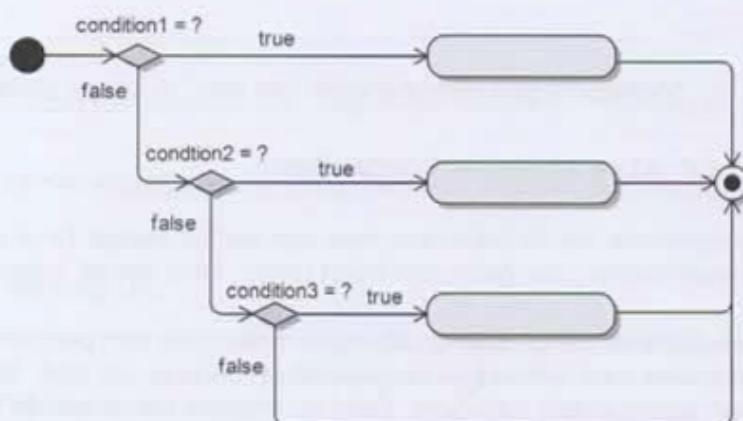
De gebruiker geeft zijn leeftijd in en krijgt op basis daarvan een verschillende boodschap.

Men kan meerdere *if else statements* in elkaar verwerken (nesten) om voor elke

voorwaarde de juiste *statements* uit te voeren:

```
if(condition1) {
    statements;
}
else{
    if(condition2) {
        statements;
    }
    else {
        if(condition3) {
            statements;
        }
    }
}
```

Schematisch:



Afbeelding 39: Schema van een geneste if-else

Dit kan men ook verkort weergeven op de volgende manier:

```
if(condition1) {
    statements;
}
else if(condition2) {
    statements;
}
else if(condition3) {
    statements;
}
```

Voorbeeld:

```
package examples.algorithms.multiselection;
import java.util.*;

public class Age {
    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);
    }
}
```



```
System.out.println("Enter your age:");
var age = keyboard.nextInt();

if (age >= 18) {
    System.out.println("You are an adult");
}
else if (age >= 10) {
    System.out.println("You are a teenager");
}
else if (age >= 2) {
    System.out.println("You are a child");
}
else {
    System.out.println("You are a baby");
}
keyboard.close();
}
```

**Opmerking:** De `? :` operator is een verkorte vorm van het `if else statement`.

#### Opdracht 14: Het `if else statement` gebruiken

- Maak een programma dat de gebruiker naar zijn leeftijd vraagt. Druk op het scherm af tot welke leeftijdsgroep de gebruiker hoort (baby, kind, tiener, volwassene).
- Maak een programma dat de BMI (*body mass index*) van een persoon berekent. Vraag de gebruiker naar zijn lengte en gewicht en bereken de BMI. Steek deze waarde in een afzonderlijke variabele. Geef vervolgens het volgende medisch advies:

lager dan 20: ondergewicht.  
tussen 20 en 25: ok.  
tussen 25 en 30: overgewicht.  
tussen 30 en 40: obesitas.  
hoger dan 40: ziekelijk overgewicht.

#### 5.5.3 Het `switch statement`

Indien meervoudige keuzes gemaakt dienen te worden op basis van een *integer*-waarde of tekenreeks (*string*), dan kan dat gebeuren met het `if else statement` uit de vorige paragraaf.

Voorbeeld: stel dat we een variabele hebben met het nummer van de maand en we willen het aantal dagen in de maand berekenen. Dan zouden we dat als volgt kunnen doen met een diep geneste `if else`:

```
int month;
int days;
...
if(month == 1)
    days = 31;
else if(month == 2)
    days = 28;
else if(month == 3)
```

```

        days = 31;
else if(month == 4)
    days = 30;
else if(month == 5)
    days = 31;
else if(month == 6)
    days = 30;
else if(month == 7)
    days = 31;
else if(month == 8)
    days = 31;
else if(month == 9)
    days = 30;
else if(month == 10)
    days = 31;
else if(month == 11)
    days = 30;
else if(month == 12)
    days = 31;
else
    days = 0;
System.out.println("Number of days in month: " + days);
    
```

Dit is een behoorlijk omslachtige constructie. Hiervoor bestaat een alternatief: het **switch case statement**.

De syntax ziet er als volgt uit:

```

switch(value) {
    case literal1:
        statements;
        break;
    case literal2:
        statements;
        break;
    ...
    default:
        statements;
}
    
```

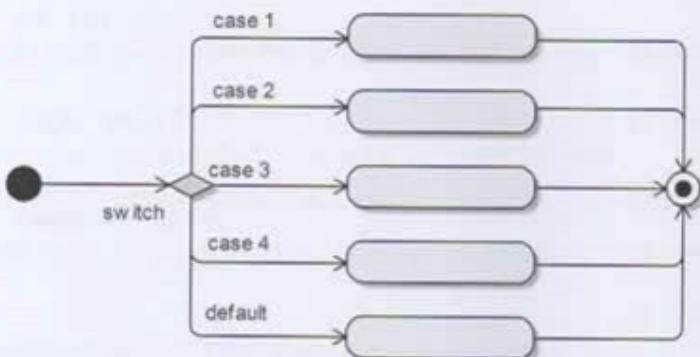
Het **switch-statement** maakt een keuze op basis van een **integer-waarde** of een waarde die automatisch kan omgezet worden in een **integer** zoals een **byte**, een **short** en een **char**. Daarnaast zijn ook tekenreeksen en opsommingstypes mogelijk. Voor elke mogelijke waarde wordt er een **case** gedefinieerd. Deze mogelijke waarde wordt aangegeven door middel van een letterlijke waarde of een constante. Dit moet steeds een waarde zijn die tijdens de compilatie gekend is (**compiletime constant**).

Na de **case** worden de **statements** gedefinieerd die uitgevoerd moeten worden als de **integer** of **string** deze waarde heeft. De **statements** worden afgesloten met een **break statement**. Indien dit niet gebeurt, zullen de **statements** van de volgende **cases** verder worden uitgevoerd tot aan de eerstvolgende **break**. Het **break statement** breekt de uitvoering op die plaats af en het programma gaat verder na de accolades van het **switch-blok**.

Bij waarden die geen expliciete **case** hebben, worden de **statements** achter **default**

uitgevoerd. De `default` case is optioneel.

Schematisch:



Afbeelding 40: Schema van een switch-case

Even een voorbeeld om dat concreet te maken. We gebruiken nu het `switch statement` om het aantal dagen in een maand te berekenen:

```
int month;
int days = 0;

switch (month) {
    case 1:
        days = 31;
        break;
    case 2:
        days = 28;
        break;
    case 3:
        days = 31;
        break;
    case 4:
        days = 30;
        break;
    case 5:
        days = 31;
        break;
    case 6:
        days = 30;
        break;
    case 7:
        days = 31;
        break;
    case 8:
        days = 31;
        break;
    case 9:
        days = 30;
        break;
    case 10:
        days = 31;
        break;
    case 11:
```

```
    days = 30;
    break;
case 12:
    days = 31;
    break;
default:
    days = 0;
}
System.out.println("Number of days in month: " + days);
```

Deze code is overzichtelijker dan die met de `if else statements`. Door de formetting wat aan te passen, kan het nog compacter:

```
int month;
int days = 0;

switch (month) {
    case 1: days = 31; break;
    case 2: days = 28; break;
    case 3: days = 31; break;
    case 4: days = 30; break;
    case 5: days = 31; break;
    case 6: days = 30; break;
    case 7: days = 31; break;
    case 8: days = 31; break;
    case 9: days = 30; break;
    case 10: days = 31; break;
    case 11: days = 30; break;
    case 12: days = 31; break;
    default: days = 0;
}
System.out.println(days);
```

De volgorde van de `cases` heeft geen belang tenzij men natuurlijk de `break` doelbewust weglaat om de `statements` van de volgende `cases` ook uit te voeren.

```
switch(integer){
    case literal1: statements;
    case literal2: statements;
    ...
    default: statements;
}
```

In dit geval zullen de `statements` vanaf de `case` uitgevoerd worden tot aan de eerstvolgende `break`. Men noemt dit ook wel een *fall through*: de code valt als het ware doorheen alle `cases` totdat een `break` de val stopt.

De code voor het berekenen van het aantal dagen in de maand kunnen we daarom ook iets compacter maken:

```
switch (month) {
    default: days = 0; break;
    case 2: days = 28; break;
```

```

    case 4:
    case 6:
    case 9:
    case 11: days = 30; break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: days = 31;
  }
  
```

We hebben hier dus verschillende *cases* waarvoor dezelfde code uitgevoerd moet worden.

### Opdracht 15: Het switch statement gebruiken

- Test bovenstaand voorbeeld uit. Vraag de gebruiker het nummer van de maand in te geven en druk het aantal dagen af. We houden geen rekening met schrikkeljaren. Druk ook een aangepaste boodschap af indien het een ongeldig nummer van de maand is.
- Maak een programma waarbij de gebruiker een letter kan ingeven. Het programma berekent de *Scrabble*-waarde van deze letter. Voor het lezen van een letter kan je het volgende gebruiken:

```

import java.io.*;

public class Scrabble {
  public static void main(String[] args) throws IOException {
    System.out.println("Enter character number");
    char c = (char) System.in.read();

    ....
  }
}
  
```

#### 5.5.4 Het while en do while statement

Een volgende stap in het programmeren is het maken van herhalingen. Hiervoor dienen onder andere het *while* en *do while statement*.

Het *while statement* voert een *statement* of codeblok uit zolang een bepaalde voorwaarde waar is. We noemen dit ook een *while-lus*.

De syntax ziet er als volgt uit:

```
while(condition)
  statement;
```

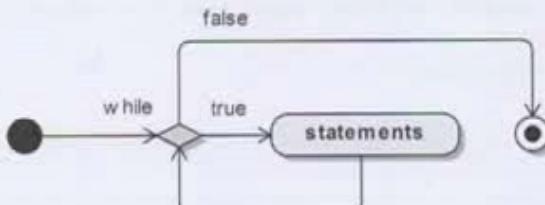
of indien meerdere *statements* moeten worden uitgevoerd:

```
while(condition) {
  statements;
```

{}

De voorwaarde is een *boolean* of een uitdrukking die als resultaat een *boolean* geeft. Als de voorwaarde *true* is, wordt het *statement* uitgevoerd. Nadien wordt de voorwaarde opnieuw geëvalueerd en als ze nog steeds *true* is, wordt het *statement* opnieuw uitgevoerd.

Schematisch:



Afbeelding 41: Schema van de while-lus

Voorbeeld:

```

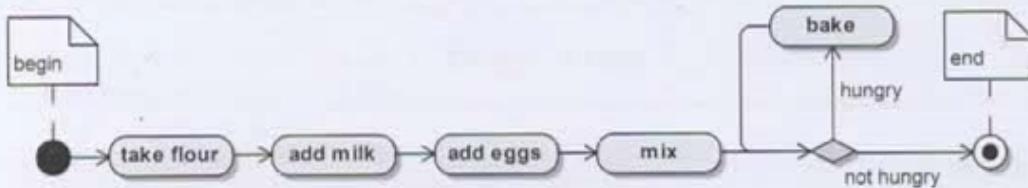
int number=1;

while(number < 10){
    System.out.println(number++);
}
    
```

In dit voorbeeld worden alle getallen van 1 tot 9 op het scherm gebracht. We gebruiken hier een variabele die we tijdens iedere herhaling verhogen. Zo'n teller gebruikt men vaak bij herhalingslussen om bij te houden hoeveel herhalingen er zijn.

We hernemen ook het voorbeeld met de pannenkoeken uit het hoofdstuk over programmatielogica.

Het schema zag er als volgt uit:



Afbeelding 42: Pannenkoeken bakken met een while-lus

De code:

```

package examples.algorithms.iteration;

public class PancakesWhile {

    public static void main(String[] args) {
        boolean hungry = true;
    }
}
    
```

```
int count = 0;

System.out.println("Take flour");
System.out.println("Add milk");
System.out.println("Add eggs");
System.out.println("Mix ingredients");

while(hungry) {
    System.out.println("Bake pancake " + ++count);
    System.out.println("Eat pancake " + count);
    if(count == 4) {
        hungry = false;
    }
}
}
```

Zolang we honger hebben, blijven we pannenkoeken bakken en eten.

Een oneindige herhaling van het codeblok maak je als volgt:

```
while(true){
    ...
}
```

Een variant van het *while statement* is het *do while statement*.

Bij het *while statement* wordt de voorwaarde geëvalueerd voordat het *statement* wordt uitgevoerd. Bij het *do while statement* wordt het *statement* eerst uitgevoerd en nadien wordt getest of het *statement* nogmaals uitgevoerd moet worden.

De syntax ziet er als volgt uit:

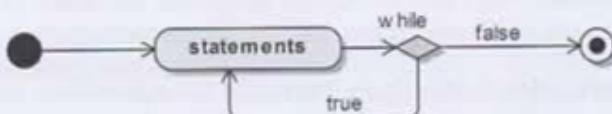
```
do
    statement;
while(condition);
```

of

```
do{
    statements;
}while(condition);
```

Het grote verschil met het *while statement* is dat het *statement* in dit geval minimaal één keer wordt uitgevoerd, ongeacht het resultaat van de voorwaarde.

Schematisch:



Afbeelding 43: Schema van de do-while-lus

Voorbeeld:

```

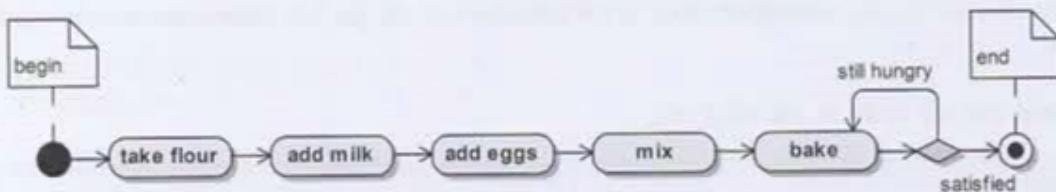
int number=1;

do{
    System.out.println(number++);
}while(number < 10);
    
```

In dit voorbeeld worden de getallen van 1 tot 9 op het scherm gebracht.

Ook hier nemen we weer het voorbeeld van de pannenkoeken.

Eerst het schema:



Afbeelding 44: Pannenkoeken bakken met een do-while-lus

En dit is de code:

```

package examples.algorithms.iteration;

public class PancakesDoWhile {

    public static void main(String[] args) {
        boolean hungry = true;
        int count = 0;

        System.out.println("Take flour");
        System.out.println("Add milk");
        System.out.println("Add eggs");
        System.out.println("Mix ingredients");

        do {
            System.out.println("Bake pancake " + ++count);
            System.out.println("Eat pancake " + count);
            if (count == 4) {
                hungry = false;
            }
        } while (hungry);
    }
}
    
```

}

Om een herhaling voortijdig te beëindigen, kan men gebruikmaken van het `break` statement. Het programma gaat dan verder na het codeblok van de herhaling.

```
while( ) {
```

```
    ...  
    break;
```

```
)
```

```
do{
```

```
    ...  
    break;
```

```
}while( );
```

Het gebruik van `break` is evenwel omstreden, omdat het erg veel weg heeft van de ongestructureerde manier van programmeren met onder andere `goto statements`. Men kan het gebruik van `break` vermijden door in de conditie van de lus het 'uitbreekscenario' op te nemen.

Met `break` ziet de code er als volgt uit:

```
while(){  
    ...  
    if(condition) {  
        break;  
    }  
    ...  
}
```

We breken de lus gewoon af bij een bepaalde voorwaarde.

Dit kunnen we beter als volgt schrijven:

```
boolean repeat = true;  
while(repeat){  
    ...  
    if(condition) {  
        repeat = false;  
    }  
    else {  
        ...  
    }  
}
```

Indien de voorwaarde voor het afbreken van de lus zich voordoet, zetten we gewoon een vlag (`repeat`) die ervoor zorgt dat de lus niet opnieuw wordt uitgevoerd. Het resultaat is hetzelfde maar de tweede manier is beter gestructureerd.

Om verder te gaan met de volgende herhaling van de lus kan men gebruikmaken van het *continue statement*. Het programma gaat verder met de controle van de voorwaarde en zal eventueel een nieuwe herhaling aanvatten.

```
while( ) {
    ...
    continue;
    ...
}
```

```
do {
    ...
    continue;
    ...
}while( );
```

Net als bij de *break* is ook de *continue* omstreden om dezelfde reden. Hier kunnen we deze constructie vermijden door de logica in de lus anders te programmeren.

We nemen nogmaals het voorbeeld. Met *continue* ziet de code er als volgt uit:

```
while() {
    ...
    if(continueCondition) {
        continue;
    }
    ...
}
```

We gaan gewoon verder met de volgende iteratie in de lus bij een bepaalde voorwaarde.

Dit kunnen we beter als volgt schrijven:

```
while() {
    ...
    if(!continueCondition) {
        ...
    }
}
```

Indien de voorwaarde van de *if* niet waar is, zullen we automatisch terechtkomen bij de volgende herhaling van de lus.

#### **Opdracht 16: Het while statement gebruiken**

- Maak een programma dat de getallen van 100 tot en met 120 afdrukt in omgekeerde volgorde.
- Maak een programma dat alle veelvouden van 3 afdrukt die kleiner zijn dan 50.
- Maak een programma dat alle machten van 5 afdrukt die kleiner zijn dan 10000.
- Maak een programma dat de letters 'A' tot en met 'Z' op het scherm afdrukt.
- Maak een programma dat de gebruiker vraagt om een getal in te geven tussen 0 en 10. Indien de gebruiker een ongeldig getal ingeeft, herhaal je de vraag.

### 5.5.5 Het **for statement** of zelfstellende lus

Bij herhalingslussen wordt vaak gebruikgemaakt van tellers die starten met een beginwaarde en bij iedere herhaling verhoogd worden.

We nemen nog even het voorbeeld van de **while-lus**:

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

Omdat dit patroon zo vaak voorkomt, is er een afzonderlijk **statement** waarmee we dit eenvoudiger kunnen doen: het **for statement**, ook wel de **zelfstellende lus** genoemd. Hierbij kunnen we op eenvoudige manier een teller definiëren die automatisch verhoogd (of verlaagd) wordt bij elke herhaling.

De syntax ziet er als volgt uit:

```
for(initialisation; condition; increment)
  statement;
```

Ook hier vervangen we het **statement** meestal door een codeblok van **statements**:

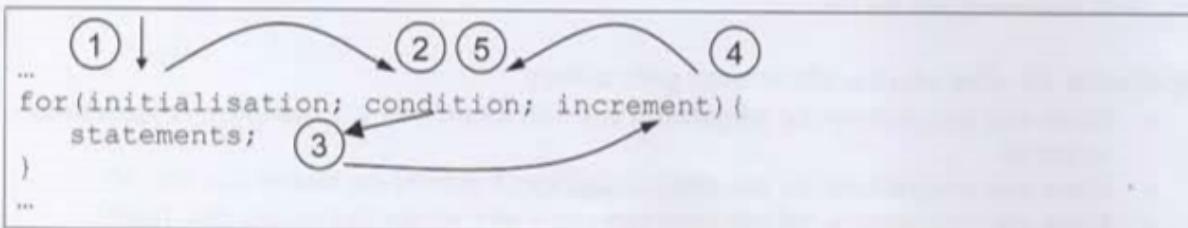
```
for(initialisation; condition; increment) {
  statements;
}
```

De **initialisation** is een uitdrukking die éénmalig wordt uitgevoerd bij het begin van de lus. Hier worden meestal de variabelen geïnitialiseerd die het aantal iteraties van de lus bepalen. Variabelen die in deze initialisatie gedeclareerd worden, hebben een bereik dat beperkt is tot het codeblok van de iteratie.

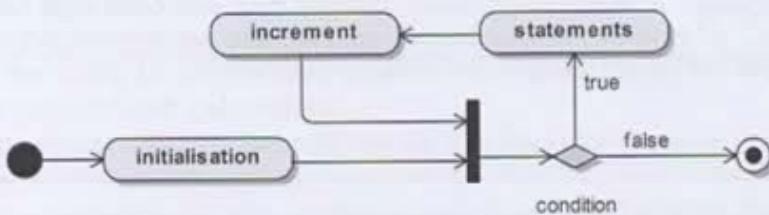
De **condition** is een uitdrukking die de voorwaarde bepaalt voor de uitvoering van de iteratie. De lus wordt uitgevoerd zolang de voorwaarde `true` is.

De **increment** is een uitdrukking die wordt geëvalueerd na elke iteratie en vóór het evalueren van de **condition**. Meestal wordt hier een teller verhoogd of verlaagd.

Het verloop ziet er als volgt uit:



Schematisch:



Afbeelding 45: Schema van de for-lus

Een voorbeeld:

```

for(int number=1; number<10; number++){
    System.out.println(number);
}
    
```

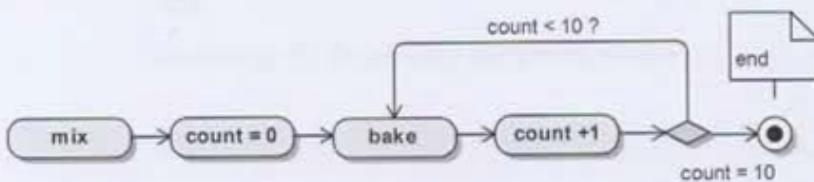
Dit voorbeeld drukt alle getallen van 1 tot 9 op het scherm.

Bij initialiseren van de teller kunnen we hier eventueel ook gebruikmaken van het type var:

```

for(var number=1; number<10; number++){
    System.out.println(number);
}
    
```

We nemen nogmaals ook het voorbeeld van de pannenkoeken. Eerst het schema:



Afbeelding 46: Pannenkoeken bakken met een for-lus

En nu de code:

```

package examples.algorithms.iteration;

public class PancakesFor {
    public static void main(String[] args) {
        System.out.println("Take flour");
        System.out.println("Add milk");
        System.out.println("Add eggs");
        System.out.println("Mix ingredients");
        for (int count = 0; count < 10; count++) {
            System.out.println("Bake pancake " + count);
            System.out.println("Eat pancake " + count);
        }
    }
}
    
```

Zowel initialisation, condition als increment zijn optioneel. Ze kunnen weggeleten

worden. Als de `condition` wordt weggelaten, komt men een oneindige lus:

```
for( ; ; ){
}
```

De `initialisation` en de `increment` kunnen bestaan uit meerdere uitdrukkingen die gescheiden worden door een komma.

```
for(i=0, j=100; i< 20 ;i++, j--){
}
```

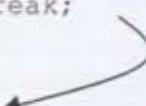
Indien men in de `initialisation` tevens de variabele declareert, geldt dit voor beide variabelen:

```
for(int i=0, j=100; i< 20 ;i++, j--){
}
```

Zowel `i` als `j` worden hierbij gedeclareerd en geïnitialiseerd.

Ook bij de `for-lus` kan men gebruikmaken van het `break` en `continue` statement om de lus af te breken of om verder te gaan met de volgende iteratie.

```
for( ; ; ){
  ...
  break;
}
...
```



```
for( ; ; ){
  ...
  continue;
}
...
```



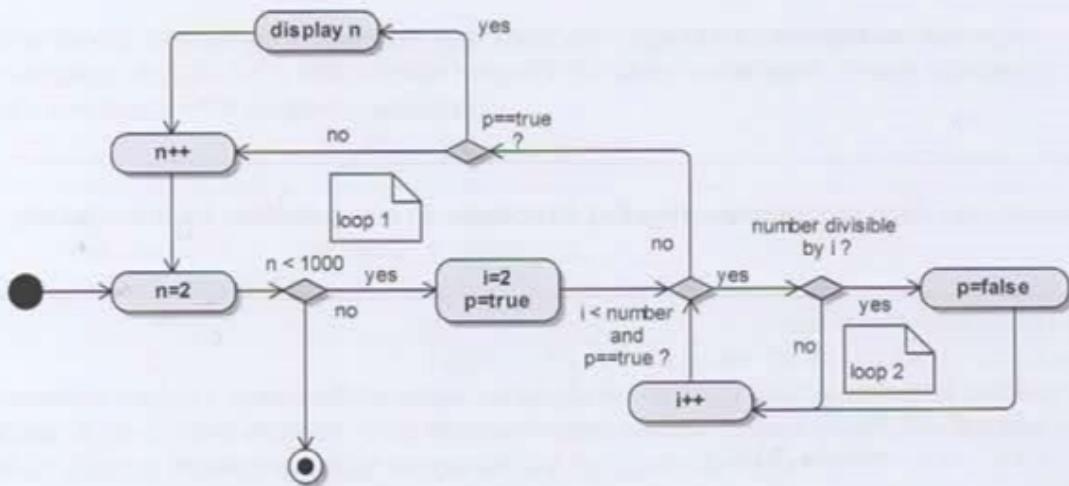
Ook hier geldt dezelfde opmerking als bij de `while` en `do while`, namelijk dat het beter is de code van lus anders te maken zodat een `break` en `continue` vermeden wordt.

Er bestaat ook een uitbreiding van de `for-lus`, namelijk de `for-each-lus`. Deze wordt gebruikt bij arrays en verzamelingen. We zullen deze lus later behandelen.

#### **Opdracht 17: Het `for` statement gebruiken**

- Maak een programma dat de getallen van 350 tot 400 afdrukt in omgekeerde volgorde.
- Maak een programma dat alle veelvouden van 7 afdrukt die kleiner zijn dan 200.
- Maak een programma dat alle machten van 11 afdrukt die kleiner zijn dan 100000.
- Maak een programma dat de letters 'z' tot en met 'a' op het scherm afdrukt.

- Maak een programma dat alle getallen afdrukt tussen  $-10$  en  $+10$ . Gebruik hiervoor een `for`-lus. Voeg bij de positieve getallen het `+`-teken toe bij het afdrukken van het getal. Het getal  $0$  heeft geen teken.
  - Maak in de vorige oefening gebruik van de `? :` operator in plaats van het `if else statement`.
  - Maak een programma dat alle getallen tussen  $0$  en  $10000$  afdrukt die zowel deelbaar zijn door  $6$  als door  $28$ . Hint: maak gebruik van de `%` operator.
  - Maak een programma dat alle priemgetallen tussen  $0$  en  $1000$  afdrukt. Druk ook het aantal gevonden priemgetallen af. Een priemgetal is een getal dat groter is dan  $1$  en enkel deelbaar is door  $1$  en door zichzelf. Hint: gebruik twee geneste `for`-lussen.
- Gebruik eventueel onderstaand schema voor het algoritme:



Afbeelding 47: Berekening van priemgetallen

## 5.6 Methoden

We hebben intussen geleerd dat programmeerregels elkaar opvolgen, en dat we ze eventueel voorwaardelijk en herhaaldelijk kunnen laten uitvoeren.

Vaak komen we in een situatie waarbij een reeks van programmeerregels telkens terugkomt. We kunnen deze programmeerregels dan groeperen en deze groep een naam geven. Vervolgens kunnen we deze groep van programmeerregels elders oproepen door de naam van de groep op te geven en eventueel extra gegevens mee te geven. Zo'n benoemde groep van programmeerregels noemen we een **methode**.

We illustreren dit met het volgende voorbeeld waarbij we gewoon de som van twee getallen berekenen:

```

public class Methods {
    public static void main(String[] args) {
        int a = 5;
        int b = 6;
        int c = a + b;
        System.out.println(c);
    }
}
    
```

Stel nu dat het berekenen van de som herhaaldelijk terugkomt in de toepassing:

```
public class Methods {
    public static void main(String[] args) {
        int a = 5;
        int b = 6;
        int c = a + b;

        System.out.println(c);

        int d = 7;
        int e = 8;
        int f = d + e;

        System.out.println(f);
    }
}
```

We kunnen dan deze programmeerregel(s) afzonderen in een methode. Dit doen we als volgt:

```
public class Methods {
    public static void main(String[] args) {
        int a = 5;
        int b = 6;
        int c = sum(a,b);

        System.out.println(c);

        int d = 7;
        int e = 8;
        int f = sum(d,e);

        System.out.println(f);
    }

    public static int sum(int number1, int number2) {
        int result = number1 + number2;
        return result;
    }
}
```

De definitie van de methode ziet er als volgt uit:

```
public static int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}
```

De woorden `static` en `public` laten we hier even buiten beschouwing. Later in de cursus zullen we hiervan de juiste betekenis zien. De naam van methode is `sum`. De programmeerregels die bij deze methode horen, zijn ondergebracht in een afzonderlijk

codeblok. Ze bevinden zich daarom tussen de twee accolades die het begin en einde van het codeblok markeren { } .

We kunnen gegevens doorgeven aan de methode. Dit noemen we parameters. We definiëren hiervoor variabelen tussen de ronde haken die volgen op de naam van de methode (int number1, int number2). Meerdere parameters worden gescheiden door een komma. Deze variabelen krijgen automatisch een inhoud zodra we de methode met haar naam oproepen:

```
int c = sum(a,b);
```

In dit geval zal de variabele number1 de waarde krijgen van de variabele a en number2 de waarde van variabele b.

Bij parameters moeten we telkens het juist datatype opgeven en kunnen we niet gebruikmaken van de var. Dat is enkel mogelijk bij lokale variabelen. In ons voorbeeld kunnen we daarom het volgende schrijven:

```
public static int sum(int number1, int number2) {
    var result = number1 + number2;
    return result;
}
```

De variabele result is een echte lokale variabele en hiervoor kan de compiler het juist datatype uit de context afleiden. Voor de parameters number1 en number2 is dat niet het geval en daarom moeten we altijd aangeven wat het juiste datatype is.

De naam van de methode hoeft niet uniek te zijn, wel de naam in combinatie met het aantal en type van de parameters. We kunnen dus twee methoden definiëren met dezelfde naam maar die een verschillend aantal parameters hebben of die parameters van een verschillend type hebben.

In het onderstaande voorbeeld hebben we drie methoden met de naam sum maar telkens met een verschillende parameterlijst:

```
public static int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}

public static int sum(int number1, int number2, int number3) {
    int result = number1 + number2 + number3;
    return result;
}

public static float sum(float number1, float number2) {
    float result = number1 + number2;
    return result;
}
```

Het definiëren van verschillende methoden met dezelfde naam maar met een verschillende parameterlijst noemt men *method name overloading*. De combinatie van de naam met de parameterlijst noemt men ook wel de 'signatuur' van de methode. Deze moet dus wel uniek

zijn.

De *compiler* zal telkens op basis van de meegegeven argumenten de juiste variant van de methode oproepen:

```
sum(1,2); -> sum(int number1, int number2)
sum(1,2,3); -> sum(int number1, int number2, int number3)
sum(1F,2F); -> sum(float number1, float number2)
```

In het codeblok van de methode kunnen we nu allerlei programmeerregels onderbrengen. Hierbij kunnen we gebruikmaken van de variabelen `number1` en `number2`.

Het resultaat van de berekening geven we terug aan de aanroeper van de methode met `return` gevuld door de berekende waarde. Het datatype van de teruggegeven waarde moeten we tevens aangeven in de definitie van de methode:

```
public static int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}
```

Doorgaans plaatsen we het *return-statement* op het einde van het codeblok, maar het is ook mogelijk dit in het midden van de code te plaatsen: de uitvoering van de methode zal dan op die plaats afgebroken worden. Dit kan nuttig zijn indien bij een complexe berekening het resultaat op een bepaald moment gekend is en men dat resultaat onmiddellijk wil teruggeven.

De teruggegeven waarde kan de aanroeper dan zelf weer gebruiken. In ons voorbeeld wordt deze waarde bewaard in de variabele `c`.

```
int c = sum(a,b);
```

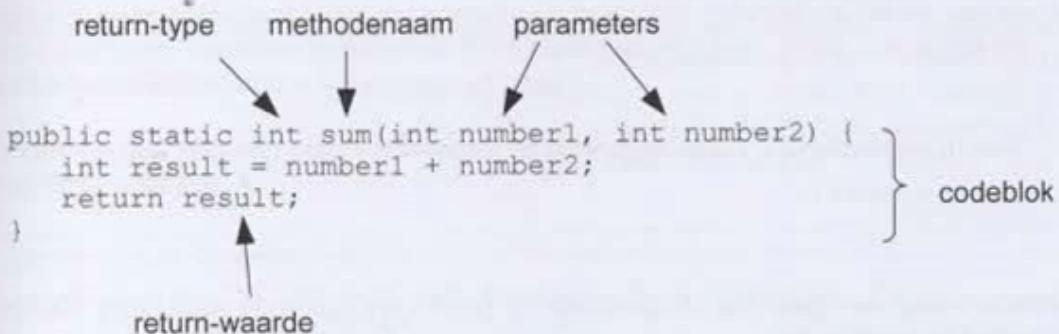
Het datatype van de *return*-waarde maakt geen deel uit van de *signatuur* van de methode. We kunnen geen twee methoden maken met dezelfde naam en parameterlijst maar wel met verschillend *return*-type.

Voorbeeld:

```
public static int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}

// Invalid !!
public static float sum(int number1, int number2) {
    int float = number1 + number2;
    return result;
}
```

We vatten de betekenis van de verschillende onderdelen nog even samen in de onderstaande tekening:



Afbeelding 48: Definitie van een methode

We kunnen nu deze methode herhaaldelijk oproepen, telkens met andere argumenten<sup>1</sup>:

```

int c = sum(a,b);
int f = sum(d,e);
int g = sum(10,13);
    
```

Deze argumenten kunnen zelf variabelen of letterlijke waarden zijn.

Indien we de waarde van een variabele meegeven, is het belangrijk te weten dat de originele variabele door de methode nooit gewijzigd kan worden. Het is enkel de inhoud van de variabele die wordt doorgegeven en terechtkomt in de parameters van de methode. Men zegt daarom dat argumenten worden doorgegeven *by value*.

Bovendien kunnen de variabelen `number1` en `number2` beschouwd worden als lokale variabelen waarvan het werkingsgebied (scope) beperkt is tot het codeblok van de methode. Om die reden is de volgende code mogelijk:

```

public class Methods {
    public static void main(String[] args) {
        int a = 5;
        int b = 6;
        int c = sum(a,b);
        System.out.println(c);
    }

    public static int sum(int a, int b) {
        int c = a + b;
        return c;
    }
}
    
```

De variabelen `a`, `b` en `c` worden op twee plaatsen gebruikt, maar omdat ze een verschillend werkingsgebied hebben dat niet conflicteert, vormt zich hier geen probleem. Het zijn wel afzonderlijke variabelen die niets met elkaar te maken hebben; toevallig hebben ze dezelfde naam.

<sup>1</sup> We spreken van 'parameters' als het gaat om de definitie van de methode en we spreken van 'argumenten' als het gaat om de gegevens die we bij het oproepen van een methode meegeven.

Methoden moeten niet noodzakelijk parameters of een *return*-waarde hebben. Een methode zonder parameters ziet er bijvoorbeeld als volgt uit:

```
public static int readNumber() {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter a number:");
    int number = keyboard.nextInt();
    return number;
}
```

Deze methode vraagt een getal aan de gebruiker en geeft vervolgens dit getal terug. De lijst van argumenten is gewoon leeg.

Verder moeten methoden ook niet noodzakelijk een waarde teruggeven. Indien ze niets teruggeven, duiden we dit aan met **void** (leeg) als *return*-type.

Bijvoorbeeld:

```
public static void printSum(int sum) {
    System.out.println("Sum: " + sum);
}
```

Indien er geen *return*-waarde is, is het *return-statement* tevens overbodig. We kunnen dit eventueel wel toevoegen als we de uitvoering van de methode onmiddellijk willen beëindigen. We gebruiken dan gewoon *return;* zonder toevoeging van een waarde.

Het geheel kan er dan als volgt uitzien:

```
public class Methods {
    public static void main(String[] args) {
        int n1 = readNumber();
        int n2 = readNumber();
        int s = sum(n1,n2);
        printSum(s);
    }

    public static int sum(int number1, int number2) {
        int result = number1 + number2;
        return result;
    }

    public static int readNumber() {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter a number:");
        int number = keyboard.nextInt();
        return number;
    }

    public static void printSum(int sum) {
        System.out.println("Sum: " + sum);
    }
}
```

We zien dat de code veel overzichtelijker is. Methoden worden namelijk niet alleen gebruikt voor code die herhaaldelijk voorkomt, maar ook om de code te structureren. In ons voorbeeld hebben we programmeerregels die samen een bepaalde activiteit uitvoeren, samengebracht in een methode. Daardoor kunnen we deze code hergebruiken, maar ook wordt het hoofdprogramma daardoor beter gestructureerd.

We kunnen hier nog een stap verdergaan en deze methoden onderbrengen in een afzonderlijke klasse.

```
public class SumUtility {
    public static int sum(int number1, int number2) {
        int result = number1 + number2;
        return result;
    }

    public static int readNumber() {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter a number:");
        int number = keyboard.nextInt();
        keyboard.close();
        return number;
    }

    public static void printSum(int sum) {
        System.out.println("Sum: " + sum);
    }
}
```

Een dergelijke klasse noemt men een *utility*-klasse of dienstklasse, omdat ze allerlei diensten in de vorm van methoden ter beschikking stelt.

We kunnen deze methoden in het hoofdprogramma dan als volgt gebruiken:

```
public class Methods {
    public static void main(String[] args) {
        int n1 = SumUtility.readNumber();
        int n2 = SumUtility.readNumber();
        int s = SumUtility.sum(n1,n2);
        SumUtility.printSum(s);
    }
}
```

Vermits de methoden nu niet langer tot de eigen klasse behoren maar tot een andere klasse, moeten we de naam van de methode laten voorafgaan door de naam van de klasse. Indien de klasse zich in een ander pakket zou bevinden, moeten we ook dit pakket importeren.

Bij complexe toepassingen kunnen we op die manier de code structureren door ze onder te brengen in afzonderlijke klassen die methoden bevatten voor een bepaalde functionaliteit, hier bijvoorbeeld voor het berekenen van de som van getallen. We noemen dit dan 'modules'. Dit is echter een techniek die gebruikelijk is bij procedurele programmeertalen. Java is evenwel een objectgeoriënteerde taal en daarin worden meer geavanceerde technieken gebruikt om afzonderlijke modules te maken. We zullen dit uitvoerig behandelen in het volgende hoofdstuk.

### Opdracht 18: Methoden

- Maak de klasse `KeyboardUtility` en `BmiUtility` zodat je ze kan gebruiken in de volgende code:

```
public class BmiApplication {  
    public static void main(String[] args) {  
        System.out.println("Enter your length (cm):");  
        int height = KeyboardUtility.readInt();  
        System.out.println("Enter your weight (kg):");  
        int weight = KeyboardUtility.readInt();  
        float bmi = BmiUtility.calculateBmi(weight, height);  
        BmiUtility.printDiagnose(bmi);  
    }  
}
```

## 5.7 Samenvatting

In dit hoofdstuk hebben we de basisregels van Java als programmeertaal geleerd. We zijn begonnen met het definiëren en gebruiken van **variabelen**. Hierin konden we gegevens tijdelijk bewaren om ze nadien opnieuw te gebruiken.

Op deze variabelen hebben we vervolgens bewerkingen uitgevoerd door middel van allerlei **operatoren**. Hiermee verkregen we dan nieuwe tussenresultaten waarmee we dan weer verder konden werken.

Vervolgens zijn we een onderscheid gaan maken tussen **uitdrukkingen**, **statements** en **codeblokken**. Deze **statements** vormen de **sequenties** in de programmatielogica.

Verder hebben we de logica uitgebreid met **keuzes** en **herhalingen**.

Ten slotte hebben we programmeerregels gegroepeerd in **methoden** en deze vervolgens ondergebracht in afzonderlijke klassen of modules.

Het gebruik van variabelen, sequenties, keuzes, herhalingen en methoden is de basis van elke programmeertaal. We hebben deze nu geleerd aan de hand van de Java-syntax.

In het volgende hoofdstuk gaan we een stap verder en bekijken we hoe we onze code op een goede manier kunnen structureren. Dit doen we met het objectgeoriënteerd programmeren.