Java traineeship

Werken met Java data types

Leerdoelen

- Maak onderscheid tussen objectreferentievariabelen en primitieve variabelen.
- Declareer en initialiseer variabelen (inclusief het casten van primitieve datatypes).
- Ontwikkel code die gebruikmaakt van wrapper-klassen zoals Boolean, Double en Integer.
- Java-operators gebruiken; inclusief haakjes om de prioriteit van de operator te overschrijven.

Primitieve variabelen

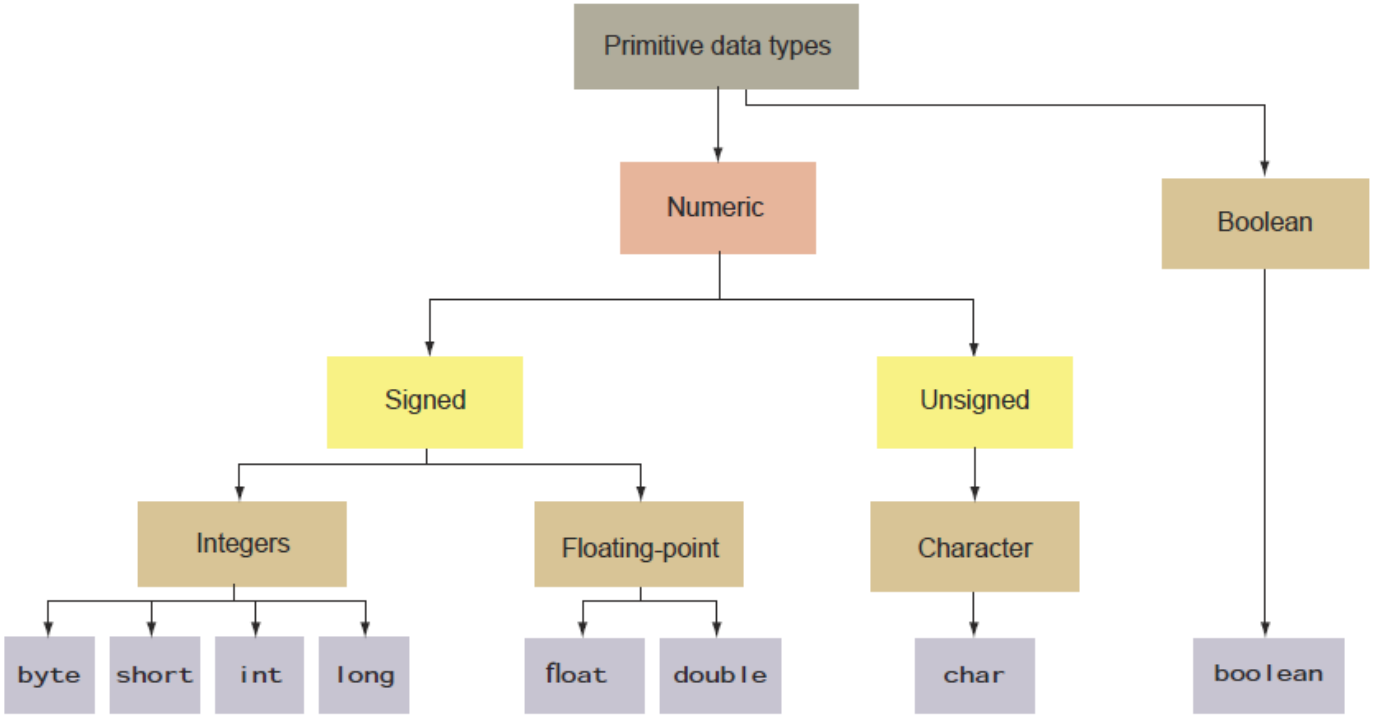
Allereerst leren we alle primitieve gegevenstypen in Java, hun letterlijke waarden en het proces van het maken en initialiseren van primitieve variabelen.

Primitieve datatypes zijn de simpelste data types in een programmeertaal.

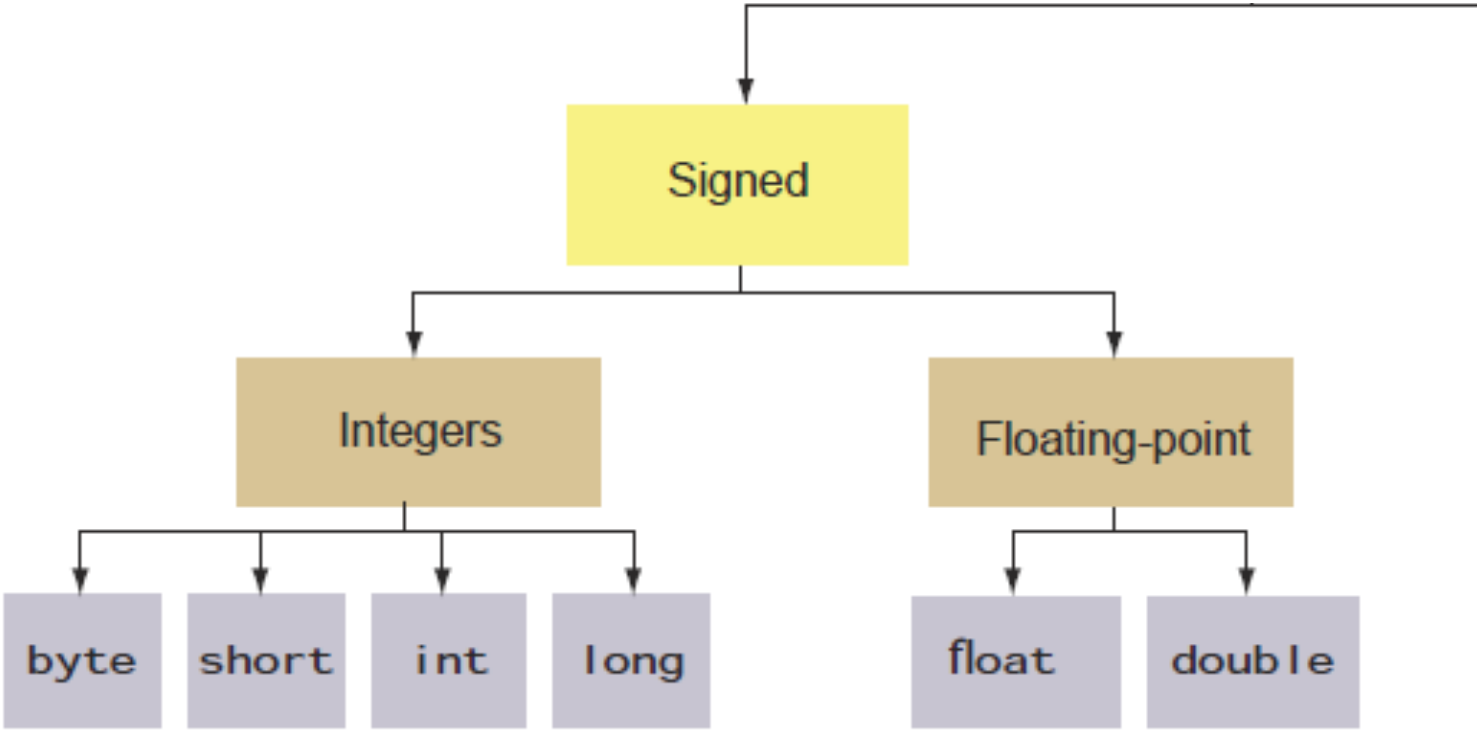
In Java heb je **acht** van deze types:
`char, byte, short, int, long, float, double, boolean`

Welke herkennen jullie al? Welke nog niet?

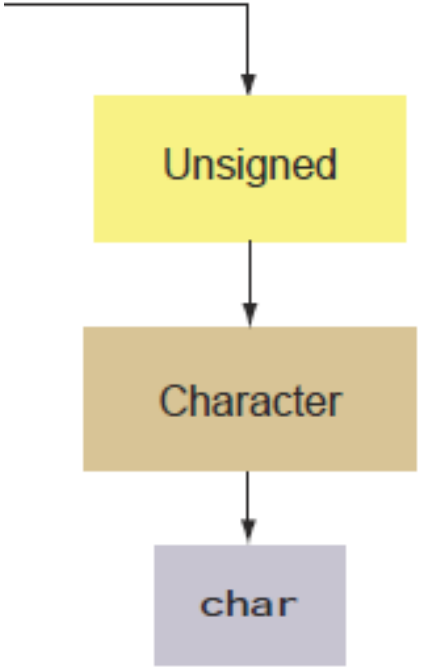
Primitieve variabelen



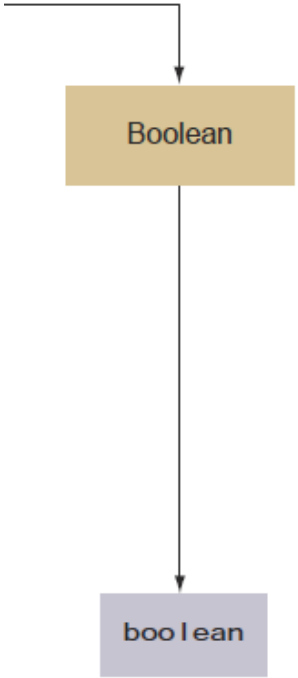
Primitieve variabelen



Primitieve variabelen



Primitieve variabelen



Categorie: Boolean

De Boolean categorie heeft maar een data type en dat is de **boolean**. Dit datatype kan slechts twee waarden bevatten: **true** of **false**

Een boolean geeft een status aan:

- Heb je vandaag je tanden gepoetst? Ja! (True)
- Heb je gestudeerd voor het tentamen? Ehmm… (False)
- …

```
boolean tandengepoetst = true;
boolean geleerd = false;
```

Categorie: signed numeric integers

Wanneer je een waarde in **gehele getallen** kan tellen, is het resultaat een geheel getal. Het bevat zowel negatieve als positieve getallen.

- Aantal Facebook vrienden
- Aantal posts op X die je vandaag hebt geplaatst
- Aantal auto’s in je straat

Categorie: signed numeric integers

Je kan de gegevenstypen **byte**, **short**, **int** en **long** gebruiken om gehele waarden op te slaan. Maar waarom zijn er meerdere typen om gehele getallen op te slaan?

Elk van deze kan een ander waardebereik opslaan. De voordelen van de kleinere liggen wellicht voor de hand: geheugenefficiëntie!

Data type	Size	Range of values
`byte`	8 bits	–128 to 127, inclusive
`short`	16 bits	–32,768 to 32,767, inclusive
`int`	32 bits	–2,147,483,648 to 2,147,483,647, inclusive
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive

Categorie: signed numeric integers

Data type	Size	Range of values
`byte`	8 bits	–128 to 127, inclusive
`short`	16 bits	–32,768 to 32,767, inclusive
`int`	32 bits	–2,147,483,648 to 2,147,483,647, inclusive
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive

Wat zou er gebeuren als we een waarde proberen op te slaan die groter is dan het datatype aan kan?

```
int i = 128;
byte b = (byte) i; // i-256 = -128
```

Wat is nu de waarde van b?

Categorie: signed numeric integers

Let op! Bytes en shorts worden verbreed naar ints. Waarom?

```
byte a = 120;
byte b = 120;
byte c = a + b; //compileerfout
```

De uitkomst van bovenstaande berekening is 240, maar de range van een byte is -128 t/m 127. We gaan dus te maken krijgen met **overflow** en daaruit ontstaat **dataverlies**.

Om zulke situaties te voorkomen kent Java het concept van **Integer Promotion**: bytes en shorts worden automatisch verbreed naar integers.

Gebruik eventueel typecasts om uitdrukkingen in een byte of short op te slaan.

Categorie: signed numeric integers

Opdracht:

Schrijf een programma dat jaren, maanden, weken, dagen en uren om kan zetten naar minuten.

Categorie: signed numeric floats

We hebben **floating-point numbers** nodig waar decimale getallen verwacht worden.

- Breuken weergeven (½ = 0.5)
- De waarde van pi weergeven (~3.14)
- Snelheid van het licht accuraat berekenen (1.079.252.848,80 kph)

Categorie: signed numeric floats

In Java kan je de gegevenstypen **float** en **double** primitieve gebruiken om decimale getallen op te slaan.

float heeft minder ruimte nodig dan double, maar kan een kleiner bereik aan waarden opslaan dan double.

float kan sommige getallen niet nauwkeurig weergeven, zelfs als ze binnen bereik zijn. Dezelfde beperking is van toepassing op double.

Data type	Size	Range of values
`float`	32 bits	+/–1.4E–45 to +/–3.4028235E+38, +/–infinity, +/–0, NaN
`double`	64 bits	+/–4.9E–324 to +/–1.7976931348623157E+308, +/–infinity, +/–0, NaN

Categorie: signed numeric floats

Hier hebben we nog wat code voorbeeldjes voor floats en doubles:

```
float gemiddelde = 20.129F;
float omtrek = 1765.65f;
double helling = 0.5;
```

Heb je het gebruik van de achtervoegsels F en f opgemerkt bij het initialiseren van de variabelen `gemiddelde` en `omtrek` in de voorgaande code?

Een decimale waarde wordt standaard als een double behandeld, maar door een ‘F’ of ‘f’ als postfix te plaatsen, vertel je de compiler dat de waarde verwerkt moet worden als een float.

Categorie: signed numeric floats

Opdracht:

Schrijf een programma dat de temperatuur van Fahrenheit omzet naar Celsius en andersom. Denk goed na over welk primitieve variabel je moet gebruiken.

Opdracht:

Schrijf een programma om de gebruiker een afstand (in meters) en de tijd te nemen (als drie getallen: uren, minuten, seconden), en geef de snelheid weer, in meters per seconde, kilometers per uur en mijlen per uur (hint: 1 mijl = 1609 meter)

```
Invoerafstand in meters: 2500
Invoeruur: 5
Invoerminuten: 56
Invoer seconden: 23
Verwachte resultaten :
Uw snelheid in meter/seconde is 0,11691531
Uw snelheid in km/u is 0,42089513
Uw snelheid in mijl/u is 0.26158804
```

Categorie: character (unsigned integer)

De character categorie definieert slechts één type: **char**.

Het kan een enkel **16-bits Unicode-teken** opslaan; dat wil zeggen, het kan karakters opslaan uit vrijwel alle bestaande scripts en talen, waaronder Japans, Koreaans, Chinees, Duits en Spaans.

Omdat je toetsenbord mogelijk geen toetsen heeft die al deze tekens vertegenwoordigen, kan je een waarde van **\u0000 (of 0)** tot een maximale waarde van **\uffff (of 65.535)** gebruiken.

De volgende code demonstreert de toewijzing van een waarde aan een char-variabele:

```
char c1 = 'D'; // let op! altijd enkele quotes voor char!!
```

Categorie: character (unsigned integer)

De oplettende kijker zag dat characters eigenlijk intern **integers** zijn. Voor de volledigheid: chars zijn **positieve** integers (unsigned).

Intern zet Java characters dus om naar getallen. Maar je kan dat ook al zelf doen:

```
char c1 = 122; //z
```

De waarde 122 is equivalent aan de letter z, zie de ASCII-tabel.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Categorie: character (unsigned integer)

Opdracht:

Caesar's Code is een van de eenvoudigste encryptietechnieken. Elke letter in de leesbare tekst wordt cyclisch vervangen door een letter op een vast aantal posities (n) in het alfabet. In deze oefening kiezen we n=3. Dat wil zeggen, 'A' wordt vervangen door 'D', 'B' door 'E', 'C' door 'F', ..., 'X' door 'A', ..., 'Z' door ' C'.

Schrijf een programma met de naam CaesarCode om de code van Caesar te coderen. Het programma zal de gebruiker vragen om een leesbare tekenreeks die alleen uit hoofdletters bestaat; bereken de cijfertekst; en druk de cijfertekst in hoofdletters af. Bijvoorbeeld,

```
Voer een leestekenreeks in: TESTING
De cijfertekstreeks is:     WHVWLQJ
```

Identifiers

Identifiers zijn namen van packages, classes, interfaces, methoden/functies en variabelen.

Vraag: Welke van de volgende coderegels wordt succesvol gecompileerd?

```
byte examen_cijfer = 7;
int lengte-in-cm = 177;
```

Objecten

Java is een object georiënteerde programmeertaal: alle functies (**methoden**) en variabelen (**attributen**) zijn onderdeel van een klasse (**class**).

Een klasse is een definitie van een datatype: een blauwdruk of bouwtekening.

Een **object** is een instantie van een klasse. (Ik of jij als individu)

Een voorbeeld: wij zijn allemaal mensen (blauwdruk), maar elk individu heeft andere attributen, aldus wij zijn allemaal verschillend (instanties).

De Punt-klasse

```
class Punt {
    double x;
    double y;
}
```

- Naam begint met een hoofdletter.
- Bevat variabelen: **class variables** en **instance variables** en functies: **methoden**.
  - Instance variablen: this.x, this.y
- Deze variabelen krijgen een nulwaarde, wanneer er geen waarde aan wordt gehangen: 0, 0.0, false of null.
  - Afhankelijk van het datatype uiteraard!
- Indien **public**, opgeslagen in een bestand met dezelfde naam.
  class Punt dus in **Punt.java**
  - Er mag maar één class public zijn per bestand.

Gebruik van deze punt klasse

```
class College {
    public static void main(String[] args) {
        Punt punt1 = new Punt();
        punt1.x = 1;
        punt1.y = 2;

        System.out.println(punt1.x + ", " + punt1.y);
    }
}
```

We maken een variabele met de naam **punt1** aan van het type **Punt**. We nemen aan dat het de class Punt boven dit stukje code staat.

Vervolgens vragen we Java een object te instantiëren m.b.v. **new**.

Daarna kunnen we de variabelen van de **instantie** van de Punt klasse aanpassen en weer uitlezen.

Methoden

Methoden zijn functies in classes:

```
class Punt {
    double x;
    double y;

    double afstandTotOorsprong() {
        return Math.sqrt(x * x + y * y);
    }
}
```

En dan kunnen we deze methode aanroepen:

```
Punt punt1 = new Punt();
punt1.x = 1;
punt1.y = 2;
System.out.println(punt1.afstandTotOorsprong());
```

Operatoren

Je hebt in de eerste presentatie al kort kennis gemaakt met enkele rekenkundige operatoren. Er zijn echter veel meer operatoren die we in Java kunnen gebruiken

We kunnen de operatoren opdelen in vier categorieën:
Toewijzend, Rekenkundig, Relationeel, Logisch

Operator type	Operators
Assignment	=, +=, -=, *=, /=
Arithmetic	+, -, *, /, %, ++, --
Relational	<, <=, >, >=, ==, !=
Logical	!, &&, ||

Assignment operators

De opdrachtoperatoren die je moet kennen voor het examen zijn =, +=, -=, *= en /=.

De = operator is een simpele toewijzingsoperator. Zoals int a = 1;

De rest van de operatoren zijn short-forms van optellen, aftrekken, vermenigvuldigen en delen.

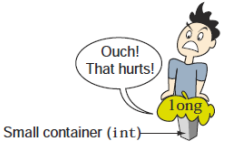
```
a -= b is gelijk aan a = a – b
a += b is gelijk aan a = a + b
a *= b is gelijk aan a = a * b
a /= b is gelijk aan a = a / b
a %= b is gelijk aan a = a % b
```

Assignment operators: compiler fouten

```
double myDouble = true; // Je kan geen boolean omzetten naar een double.
boolean b = 'c'; // Je kan geen char omzetten naar een boolean.
boolean b1 = 0; // Je kan geen int omzetten naar een boolean.
boolean b2 -= b1; // je kan boolean met boolean niet van elkaar aftrekken.
```

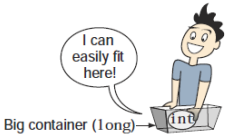
Laten we nu proberen de variabelen die een groter bereik aan waarden kunnen opslaan in variabelen met een korter bereik te persen.

```
long num = 100976543356L;
int val = num; // Deze waarde is te groot voor een int.
```



Je kan nog steeds grotere waarden in kleinere variabelen stoppen, maar deze waarde zal waarschijnlijk niet overeen komen met de oude waarde.
Andersom werkt altijd!

```
int intVal = 1009;
long longVal = intVal;
```



Rekenkundige operatoren

Operator	Doel	Gebruik	Antwoord
+	optellen	12 + 10	22
-	aftrekken	19 - 29	-10
*	vermenigvuldigen	101 * 45	4545
/	Delen	10 / 6 10.0 / 6.0	1 1.66666666666667
%	modulus	10 % 6 10.0 % 6.0	4 4.0
++	Huidige waarde optellen met 1	++var of var++	11 (ervan uitgaand dat var 10 was)
--	Huidige waarde aftrekken met 1	--var of var--	9 (ervan uitgaand dat var 10 was)

Rekenkundige operatoren: chars

Weet je nog? Chars worden gerepresenteerd door nonnegatieve getallen (0 of groter) en omgezet naar karakters middels de ASCII tabel.

```
char char1 = 'a';
System.out.println(char1); // geeft a terug
System.out.println(char1 + char1); // geeft 194 terug
```

Increment: prefix en postfix notatie

Om een variabele met exact 1 te verhogen of te verlagen zijn de **++** (increment) en **--** (decrement) operatoren in veel programmeertalen aanwezig.

```
int i = 2;
i++;
System.out.println(i); // 3
++i;
System.out.println(i); // 4
```

Maar wat is dan het verschil?

Increment: prefix en postfix notatie

Op de oppervlakkige laag doen i++ en ++i hetzelfde: ze verhogen i met exact 1.

Bij gebruik in expressies, is er toch een verschil:

- i++: i wordt verhoogd met 1 **ná** de expressie.
- ++i: i wordt verhoogd met 1 **vóór/tijdens** de expressie.

```java
int i = 2, j = 2;

System.out.println(i++ + 3); // 5
System.out.println(i);       // 3
System.out.println(++j + 3); // 6
System.out.print(j);         // 3
```

Relationele operatoren

Relationele operatoren worden gebruikt om een voorwaarde te controleren o.b.v. een vergelijking.

Relationele operatoren kan je verdelen in twee categorieën:

- Vergelijken of waarden **groter** (>, >=) of **kleiner** (<, <=) zijn
- Vergelijken van waarden voor **gelijkheid** (==) en **ongelijkheid** (!=)

Relationele operatoren

De operatoren <, <=, > en >= werken met numerieke getallen, zowel signed als unsigned.

```
int i1 = 10;
int i2 = 20;
System.out.println(i1 >= i2); // false
long long1 = 10;
long long2 = 20;
System.out.println(long1 <= long2); // true
```

Relationele operatoren

De operatoren == (gelijk aan) en != (niet gelijk aan) kunnen worden gebruikt om alle primitieve datatypes te vergelijken.

De operator == retourneert de boolena waarde true als de primitieve waarden gelijk zijn, en anders false.

De operator != retourneert true als de primitieve waarden niet gelijk zijn, en anders false.

Volg je het nog?

Relationele operatoren

```java
int a = 10;
int b = 20;
System.out.println(a == b);      // false
System.out.println(a != b);      // true
boolean b1 = false;
System.out.println(b1 == true);  // false
System.out.println(b1 != true);  // true
System.out.println(b1 == false); // true
System.out.println(b1 != false); // false
```

Logische operatoren

1. conditionele and: **&&**
2. conditionele or: **||**
3. bitwise and: **&**
4. bitwise o:r **|**
5. 3 en 4 afgekort: **&=** en **|=**

AND, OR en NOT operaties

Operators && (AND)	Operator || (OR)	Operator ! (NOT)
true && true → true true && false → false false && true → false false && false → false true && true && false → false	true || true → true true || false → true false || true → true false || false → false false || false || true → true	!true → false !false → true

- Logical AND (&&)
  - Evalueert naar *true* als beide stellingen true zijn, anders *false*
- Locical OR (ll)
  - Evalueert naar *true* als één of alle stellingen *true* zijn, anders false
- Locical NOT (!)
  - Evalueert *true* naar *false* en vice versa.

Short-circuit

Als Java && tegenkomt en de linkerstelling genereert false, dan wordt de rechterexpressie niet verwerkt.

```
public class Test {
    public static void main(String[] args) {
        boolean d;
        d = (2 > 3) && (3 > 2); // d is false
    }
}
```

De expressie 3 > 2 zal nooit worden geëvalueerd omdat 2 > 3 false oplevert.

Short-circuit

Als Java een || operator tegenkomt en de linkerstelling genereert true, dan wordt de rechterstelling niet verwerkt.

```java
public class Test {
    public static void main(String[] args) {
        int n = 0, m = 1;
        boolean b;
        b = (n == 0) || (m/n > 2);
        System.out.println("b is" + b);
        System.out.println("m/n is" + m/n);
    }
}
```

```
> java Test
b is True
Exception in thread “main”
java.lang.ArithmeticException
/ by zero
At Test.main(Test.java:9)
```

Bitwise operators en boolean

Bitwise operators maken geen gebruik van short-circuit, dus elke stelling wordt geëvalueerd.

Delen door 0 kan niet, dus ERROR voor de volledige operation, ondanks dat het eerste deel wel waar is.

```
public class Test2 {
    public static void main(String[] args) {
        int n = 0, m = 1;
        boolean b;
        b = (n == 0) | (m/n > 2);
        System.out.println("b is" + b);
        System.out.println("m/n is" + m/n);
    }
}
```

```
> java Test2
Exception in thread “main”
java.lang.ArithmeticException
/ by zero
At Test.main(Test.java:9)
```

Bitwise operators en integer

```
public class BitDemo {
    public static void main(String[] args) {
        int x, y, result;
        x = 50;           // 0011 0010
        y = 51;           // 0011 0011
        result = x & y;   // 0011 0010
        System.out.println("result of x & y is " + result);
        // result of x & y is 50

        int a = 1;        // 0001
        a |= 4;           // 0001 | 0100 = 0101
        System.out.println("a = " + a); // a = 5
    }
}
```

Bitwise AND:
Als beide bits 1 zijn, dan zal de operation 1 teruggeven, anders 0.

Bitwise OR: Als minimaal één van de twee bits 1 is, dan zal de operation 1 teruggeven, anders 0.

Overzicht shortcut operatoren voor booleans

En ook voor booleans is een verkorte notatie ingebouwd:

```
boolean b = false

b = b | true;
// of
b |= true; // denk aan a += b

b = b & true;
// of
b &= true;
```

Operator precedence

Wat gebeurt er als je meerdere operators gebruikt binnen een enkele regel code?

Welke operators hebben voorrang op andere?

Operator precedence

Operator	Precedence
Postfix	`Expression++, expression--`
Unary	`++expression, --expression, +expression, -expression, !`
Multiplication	`*` (multiply), `/` (divide), `%` (remainder)
Addition	`+` (add), `-` (subtract)
Relational	`<, >, <=, >=`
Equality	`==, !=`
Logical AND	`&&`
Logical OR	`||`
Assignment	`=, +=, -=, *=, /=, %=`

Operator precedence

```
int int1 = 10, int2 = 20, int3 = 30;
System.out.println(int1 % int2 * int3 + int1 / int2);
```

Wat levert dit op?

```
(((int1 % int2) * int3)) + (int1 / int2)
(((10 % 20) * 30)) + (10 / 20)
( (10 * 30)) + (0)
( 300 )
```

```
int int1 = 10, int2 = 20, int3 = 30;
System.out.println(int1 % int2 * (int3 + int1) / int2);
```

En nu?

Wrapper classes

Primitieve types zijn leuk en aardig, maar soms niet zo handig

```
String mijnZin = 5.toString();
ArrayList<int> mijnLijstje; // Generiek type moet non-primitief zijn.
```

1. Methoden aanroepen op primitieve datatypen is niet mogelijk.
2. Generics in Java accepteren geen primitief datatype

Hoe kunnen we dit dan alsnog voor elkaar krijgen? Wrappers!!

Oplossing

Java biedt zogenaamde **wrapper** klassen aan voor de primitieve types. Dit zijn klassen die extra functionaliteit bieden, waardoor je impliciet functies kunt aanroepen op primitieve datatypen.

Class name	Method
Boolean	`public static boolean parseBoolean(String s)`
Character	`no corresponding parsing method`
Byte	`public static byte parseByte(String s)`
Short	`public static short parseShort (String s)`
Integer	`public static int parseInt(String s)`
Long	`public static long parseLong(String s)`
Float	`public static float parseFloat(String s)`
Double	`public static double parseDouble(String s)`

Voordelen

Een voordeel van wrappers is dat ze allerlei handige methoden bevatten, waarvan de meeste statisch zijn. Dat wil zeggen dat methoden tot een class behoren en niet tot een object, en worden daarom ook op een class aangeroepen en niet een object.

```
System.out.println(new Integer(5).toString()); // Prints: 5
System.out.println(Integer.toString(5));       // Prints: 5
System.out.println(Integer.toString(5, 2));    // Prints: 101
System.out.println(Integer.max(7, 18));        // Prints: 18
System.out.println(Integer.parseInt("5") + 6); // Prints: 11
```

Over het algemeen heeft het handmatig aanmaken van *Integer* variabelen weinig nut.

Voordelen

Daarnaast ook handig voor bijvoorbeeld ArrayLists:

```
ArrayList<int> mijnLijstje;     // Dit mag niet. :(
ArrayList<Integer> mijnLijstje; // Dit mag wel! :D
```

Nadelen

Dit maakt je code wel minder leesbaar, omdat je constant nieuwe objecten moet introduceren en niet meer gewoon mag zeggen

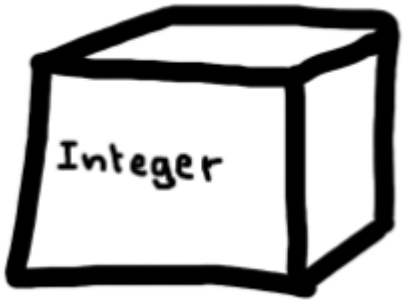
```
Integer a = 5;
```

Of toch wel? :O

Autoboxing

Java maakt het je makkelijk door primitieven automatisch om te zetten naar hun wrapper indien nodig. Dat noemen we **autoboxing**

```
Integer a = 5; // dit mag toch wel!
```



ints toevoegen aan een ArrayList van *Integers* werkt ook prima, want Java stopt ze automatisch in een doosje voor je.

Unboxing

Dit werkt ook de andere kant op:

Integers kunnen automatisch uitgepakt worden naar *ints*. De noemen we **unboxing.**

```
int i = a; // 5
```

Dit werkt wederom ook bij het ophalen vanuit een ArrayList. Hier in een ander hoofdstuk meer over…

Wrapper classes

Moraal van het verhaal: door **autoboxing** en **unboxing** kun je gewoon **primitieve types blijven gebruiken** en heb je toch het voordeel van geavanceerde Java functionaliteit zoals **ArrayLists**!

Eindopdracht:

In deze opdracht moet via de terminal **een geheel positief getal** opgegeven worden.
Het getal moet opgevraagd worden met behulp van de **nextInt-methode** uit de **Scanner-klasse**.

Zorg ervoor dat verkeerde invoer goed wordt afgehandeld door het programma.

Vervolgens moet de som van alle even getallen van 1 tot en met het opgegeven getal worden berekend.

Ook moet de som van alle oneven getallen van 1 tot en met het opgegeven getal worden berekend.
Tip: controleer of het getal een even getal is met de %.

Als laatste moet het verschil van deze twee sommen worden geprint.

Een voorbeeld van de uitvoer is:

```
$ java Opdracht2
Geef een geheel positief getal: 10
som van oneven getallen tot en met 10 is 25
som van even getallen tot en met 10 is 30
verschil tussen twee sommen is -5
```

Leerdoelen

- ✓ Maak onderscheid tussen objectreferentievariabelen en primitieve variabelen.
- ✓ Declareer en initialiseer variabelen (inclusief het casten van primitieve datatypes).
- ✓ Ontwikkel code die gebruikmaakt van wrapper-klassen zoals Boolean, Double en Integer.
- ✓ Java-operators gebruiken; inclusief haakjes om de prioriteit van de operator te overschrijven.

Vragen?

- E-mail mij op joris.vanbreugel@code-cafe.nl!
- Join de Code-Café community op Discord!