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

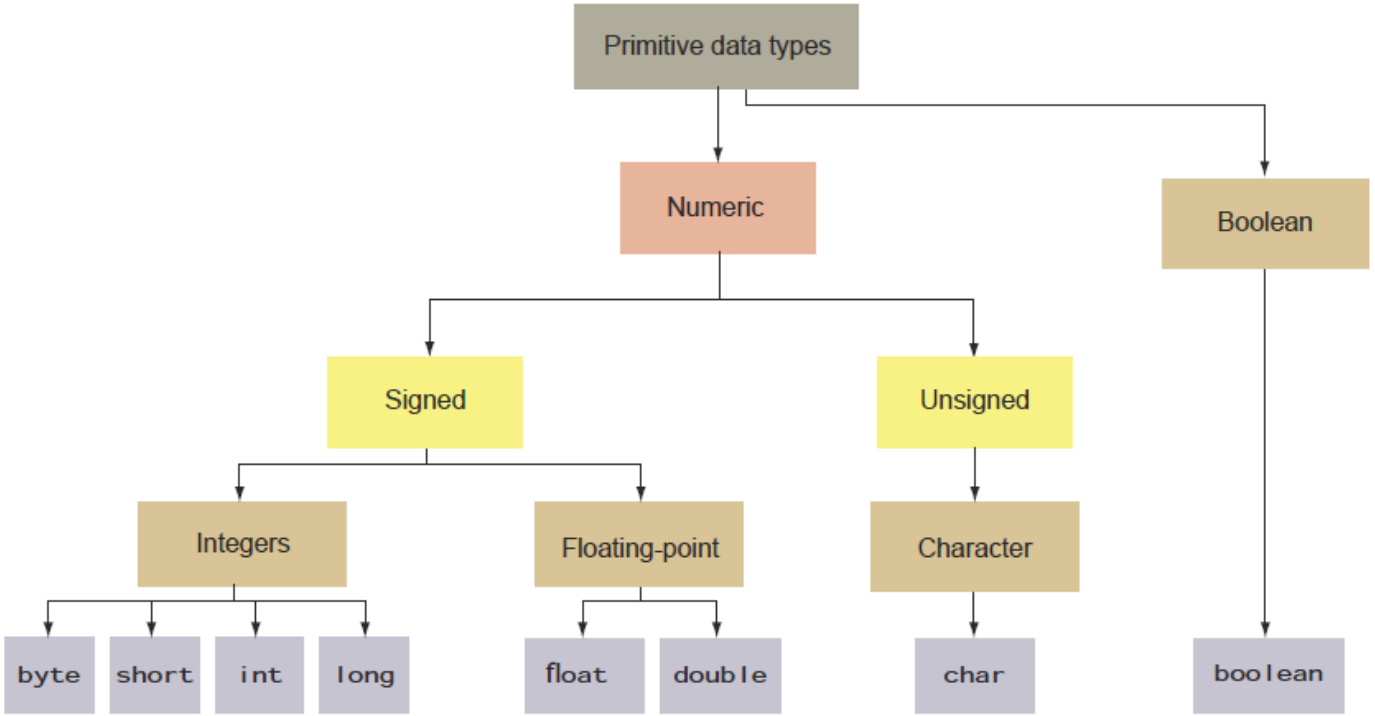
Allereerst leren we alle primitieve gegevenstypen in Java, hun letterlijke waarden en het proces van het maken en initialiseren van primitieve variabelen.

Primitieve data types 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



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 wordt gebruikt in een scenario waar twee staten bestaan:

- Heb je vandaag je tanden gepoetst?
- Heb je gestudeerd voor het tentamen?
- …

```
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 Tweets die je vandaag hebt geplaatst
- Cijfer voor het tentamen waar je niet voor had geleerd

Categorie: signed numeric integers

Je kan de gegevenstypen **byte**, **short**, **int** en **long** gebruiken om gehele waarden op te slaan. Wacht even: waarom heb je zoveel typen nodig om gehele getallen op te slaan?

Elk van deze kan een ander waardebereik opslaan. De voordelen van de kleinere liggen voor de hand: ze hebben minder geheugen nodig en zijn sneller om mee te werken.

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;
```

Wat is nu de waarde van b?

Categorie: signed numeric integers

Opdracht:

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

Denk goed na over het datatype dat je kiest. Wees zo efficiënt en realistisch mogelijk.

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**. **Floats** zijn minder nauwkeurig dan dubbel.

float kan sommige getallen niet nauwkeurig weergeven, zelfs als ze binnen bereik zijn. Dezelfde beperking is van toepassing op **dubbel**, zelfs als het een gegevenstype is dat meer precisie biedt.

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 tijdens het initialiseren van de variabelen `gemiddelde` en `omtrek` in de voorgaande code?

Het standaardtype van een decimale letterlijke waarde is double, maar door een decimale letterlijke waarde met F of f als achtervoegsel te plaatsen, vertel je de compiler dat de letterlijke waarde moet worden behandeld als een float en niet als een double.

Hetzelfde is mogelijk met doubles. Voeg dan een d of een D toe aan het einde van je variabel.

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 gegevenstype: **char**.
Een char is een geheel getal zonder teken.

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 toont 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. Welke waarde equivalent is aan welke letter kunnen we opzoeken in het ASCII-tabel

Table of ASCII characters 0-127 (x0000-x007f)

000	--0	--1	--2	--3	--4	--5	--6	--7	010	--0	--1	--2	--3	--4	--5	--6	--7
00-	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	10-	@	A	B	C	D	E	F	G
01-	BS	HT	LF	VT	FF	CR	SO	SI	11-	H	I	J	K	L	M	N	O
02-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	12-	P	Q	R	S	T	U	V	W
03-	CAN	EM	SUB	esc	FS	GS	RS	US	13-	X	Y	Z	[	\	]	^	_
04-		!	"	#	$	%	&	'	14-	`	a	b	c	d	e	f	g
05-	(	)	*	+	,	-	.	/	15-	h	i	j	k	l	m	n	o
06-	0	1	2	3	4	5	6	7	16-	p	q	r	s	t	u	v	w
07-	8	9	:	;	<	=	>	?	17-	x	y	z	{	|	}	~	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

Reminder: Wat waren identifiers ook al weer?

Identifiers zijn namen van pakketten, klassen, interfaces, methoden 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 zijn onderdeel van een klasse (**class**) of een **object**.

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

Een object is een instantie van een klasse.

Laten we dit nader bekijken met wat voorbeelden!

De punt klasse

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

- Naam begint met een hoofdletter
- Bevat variabelen: **class variables** en **instance variables** en functies: **methoden**.
- Deze variabelen krijgen de “standaard” waarden: 0, 0.0, false of null.
- Indien **public**, opgeslagen in een bestand met dezelfde naam.
  class Punt dus in **Punt.java**

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**.

Vervolgens vragen we Java een object te instanti¨eren met behulp van het new keyword.

Daarna kunnen we de variabelen van de **instantie** van de Punt klasse aanpassen en weer uitlezen. Zo'n instantie wordt ook een **object** genoemd.

Methoden

Klassen kunnen ook methoden bevatten, dat ziet er als volgt uit:

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

    double aftandTotOorsprong() {
        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.aftandTotOorsprong());
```

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.

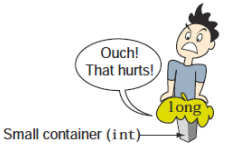
```
a -= b is gelijk aan a = a – b
a += b is gelijk aan a = a + b
a *= b is gelijk aan a = a * b
a /= b is gelijk aan a = a / b
a %= b is gelijk aan a = a % b
```

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.
```

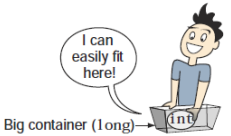
Laten we nu proberen de variabelen die een groter bereik aan waarden kunnen opslaan in variabelen met een korter bereik te persen.

```
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 getallen 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 **++** en **--** operatoren in veel programmeertalen aanwezig

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

Increment: prefix en postfix notatie

Wanneer gebruikt als statement, doen zowel **i++** als **++i** hetzelfde.

Echter, als je ze gebruikt als (onderdeel van een) expressie wordt bij ++i eerst de variabele opgehoogd alvorens de waarde wordt gebruikt in de berekening, en bij i++ de waarde gebruikt en aan het einde van de expressie pas opgehoogd:

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

Relationele operatoren

Relationele operatoren worden gebruikt om één voorwaarde te controleren. U kunt deze operators gebruiken om te bepalen of een primitieve waarde gelijk is aan een andere waarde of kleiner of groter is dan de andere waarde.

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 alle soorten getallen, zowel gehele getallen (inclusief char) als drijvende komma, die kunnen worden opgeteld en afgetrokken.

```
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 soorten primitieven te vergelijken: **char, byte, short, int, long, float, double en boolean**.

De operator == retourneert de booleaanse 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.

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 **&&** voor booleans
2. conditionele or **||** voor booleans
3. conditionele and **&** voor booleans
4. conditionele or **|** voor booleans
5. bitwise and **&** voor integer typen
6. bitwise or **|** voor integer typen
7. verkorte **&=** 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 alle operatoren true zijn, anders *false*
- Locical OR (ll)
  - Evalueert naar *true* als een of alle operanden *true* zijn, anders false
- Locical NOT (!)
  - Evalueert *true* naar *false* en vice versa.

Logische operatoren && en ||

Als Java && operator of || tegenkomt, wordt expressie rechts van operator niet geëvalueerd als dat niet nodig is.

```
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.

Delen door 0 in tweede boolean expressie niet uitgevoerd

Als Java && operator of || tegenkomt, wordt expressie rechts van operator niet geëvalueerd als dat niet nodig is.

```
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)
```

Evalueren van beide boolean expressies met & en |

```java
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 operatoren op integer typen

```
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
    }
}
```

Overzicht shortcut operatoren voor booleans

En ook voor booleans is een verkorte notatie ingebouwd:

```
boolean b = false

b = b | true;
// of
b |= true;

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

Operator precedence

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

Welke moet als de koning worden behandeld en de voorkeur krijgen boven de anderen?

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(); // Primitief type heeft geen methoden.
ArrayList<int> mijnLijstje; // Generiek type moet non-primitief zijn.
```

Hoe kunnen we dit nou alsnog voor elkaar krijgen?

Oplossing

Java biedt zogenaamde **wrapper** klassen aan voor de primitieve types. Dit zijn klassen die niets anders bevatten dan het primitieve type waar ze bij horen, maar die wel extra functionaliteit bieden.

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 (wat betekent dat ook al weer?)
Zodat je niet eens een wrapper object nodig hebt.

```
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
```

Mag je nou nooit meer *int* gebruiken? 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

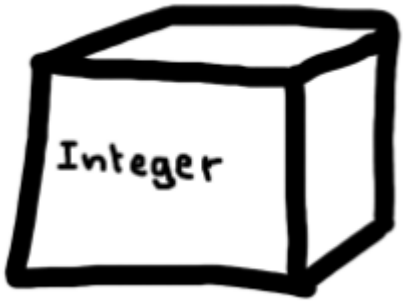
Dit maakt je code wel minder leesbaar, omdat je allemaal nieuwe klassenamen gaat introduceren en niet meer gewoon mag zeggen

```
Integer a = 5;
```

Of toch wel? ;)

Boxing

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!
```

En *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 dan weer **unboxing**

```
int i = new Integer(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. 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 voornaam.achternaam@code-cafe.nl!

- Join de Code-Café community op discord!