Kapittel 2:
Grunnleggende programelementer

Redigert av:

Khalid Azim Mughal (`khalid@ii.uib.no`)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.
ISBN: 82-02-24554-0
`http://www.ii.uib.no/~khalid/jfps3u/`
(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

- Programmeringsspråkets oppbygging
- Tallsystemer
- Primitive datatyper
- Variabler
- Uttrykk og operatorer
- Innlesning av verdier fra terminalen
- Formatert utskrift
- Programutviklingsprosess
- Valgsetninger: `if` og `if`-else
- Algoritmeutvikling — pseudokode
- Løkker: `while` og `do-while`
- Påstander: `assert`-setning

Et språk = syntaks + semantikk

Naturlige språk | **Programmeringsspråk**

SYNTAKS | **SYNTAKS**

SEMANTIKK | **SEMANTIKK**

Hele den store verden

vokabular

"lovlige" ord:
lege, en, jeg, er, som, han ...

identifikatorer, litteraler, spesielle tegn:
`public, sum, 10,'0',"tast inn"...`

fraser

sammensatte fraser:
en lege, som kom i går ...

uttrykk:
`sum+1, sum/antall ...`

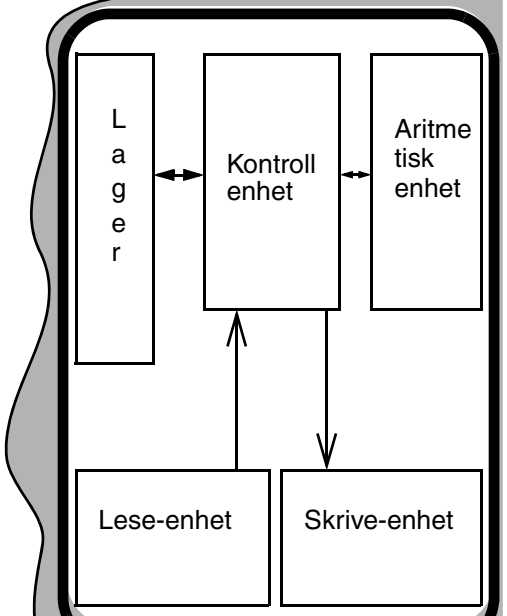
setninger

kategorier: *substantiv, verb ...*
substantiv + verb
Jeg er, En lege går, Huset flyr

setningstyper:
påstand
ordre

kategorier: *variabel, uttrykk ...*
variabel = uttrykk
`sum = sum+1`

imperativer:
tilordning (lager)
kontroll



Datamaskinens "språk": binært tallsystem

- Desimalt tallsystem (base: `10`, siffer: `0-9`)
- *Binært tallsystem (base:* `2`, *siffer:* `0` *og* `1`*) - brukes til å representere informasjon i datamaskinen.*
- Oktalt tallsystem (base: `8`, siffer: `0-7`)
- Hexadesimalt tallsystem (base: `16`, siffer: `0-9` og `A-F`, der `A=10,B=11...,F=15`)

Desimalt:	Binært:	Oktalt:	Hexadesimalt:
`29`	`00011101`	`35`	`1D`

Se også vedlegg E.

Konvertering mellom tallsystemer

- Konvertering fra binært (base = 2) til desimalt tallsystem:

$$
\begin{aligned}
11101_2 &= 2^4*1 + 2^3*1 + 2^2*1 + 2^1*0 + 2^0*1 \\
&= 16 + 8 + 4 + 0 + 1 \\
&= 29_{10}
\end{aligned}
$$

- Konvertering fra desimalt til binært tallsystem:

$$
\begin{aligned}
29 &= 14*2 + 1 \\
14 &= 7*2 + 0 \\
7 &= 3*2 + 1 \\
3 &= 1*2 + 1 \\
1 &= 0*2 + 1
\end{aligned}
$$

$$29_{10} = 11101_2$$

Konvertering mellom tallsystemer (forts.)

- Konvertering fra binært til oktalt tallsystem (3-bits grupper fra høyre mot venstre):

Binært:	00	011	101
Oktalt:	0	3	5

- Konvertering fra binært til hex tallsystem (4-bits grupper fra høyre mot venstre):

Binært:	0001	1101
Hex:	1	D

- Regning med *negative* tall:

```
 29      00011101          00011101     00011101
-29   -(00011101)     (11100010 + 1)   +11100011
----                                   ---------
  0                                     00000000
```

toer-komplement: = *ener-komplement + 1*

- Tall representeres med *fast antall bits* i lager, dette medfører begrensninger i tall-representasjonen!

  $\frac{2}{3}$ = `0.666...` kan *ikke* representeres eksakt i lager!

Identifikatorer og litteraler

Identifikator

- Navn i et program kalles for *identifikatorer*.
- En identifikator er en sekvens av tegn som er enten *bokstav, siffer, understrek* (_) eller *dollar-tegn* ($), og kan ikke begynne med et siffer.
- Identifikatorer med små og store bokstaver er forskjellige (*case-sensitive*).

definerte: `Antall, antall, sum_$, maks_sum, bingo_, teller, $$_100`
– deklarerte i programmet av programmereren.

nøkkelord: `float, if, true, while, for`
– reserverte ord med fast betydning i språket.

Eksempel på ulovlige identifikatorer: `hva?, 7UP`

Litteral

- betegner en konstant verdi

heltall: `59 0 1993 -46`

flyttall: `-3.1 .5 0.5 5E-1`(=5×10^{-1}=0.5)

tegn: `'a' '0' '+' '5' '\n' '\u000a'`

boolsk: `true false`

streng: `"a piece of cake" "if" "\n" "0" "3.14"`

I tillegg har man også *spesielle tegn*, som `( ) { } [ ] ; .`
og *operatorer*, f.eks. `+ * - % = == != >=`

Grunnleggende enheter

- Identifikatorer, litteraler, operatorer og spesialtegn utgjør det som kalles for *grunnleggende enheter* (*tokens*).
- Et program på en fil er en lang sekvens av tegn.
- Kompilatoren må hakke programmet opp i grunnleggende enheter for å oversette programmet.
- I enkelte tilfeller er det nødvendig å skille grunnleggende enheter i programmet med *hvite blanke* (*white spaces*: sekvens av *mellomrom*, *tabulator*, *linjeskift* (*newline*) og *form-feed*):

```
return 0;  // må skrives med et blanke tegn mellom return og 0
return0;   // ulovlig setning som består av en enkel identifikator
return0.   // ulovlig

sum=sum+1; // ikke nødvendig å skille grunnleggende enheter
```

- Hvite blanke kan forbedre *lesbarhet*!

Programdokumentering v.h.a. kommentarer

- Kommentarer i Java kan skrives i tre varianter.

```
// Denne kommentaren går til slutten av linjen.
/*
   Denne kommentaren går over
   flere linjer.
 */

/** (NB. Ekstra stjerne.)
 * Denne kommentaren må forekomme før en klassedeklarasjon,
 * klassemedlem eller klassekonstruktør, og blir
 * inkludert i automatisk generert dokumentasjon vha javadoc.
 **/
```

- Kommentarer kan ikke nøstes.
- Generelt kan en kommentar forkomme der en hvit blank kan forekomme i programmet.
- Kommentarer og hvite blanke blir (stort sett) ignorert av kompilatoren.

Data

- Java program manipulerer to typer data:

1. *primitive verdier*
2. *referanseverdier* til *objekter*.

- Forståelse av disse to typene av data er veldig viktig for å tolke resultatet av enkelt operasjoner/ beregninger/ handlinger!

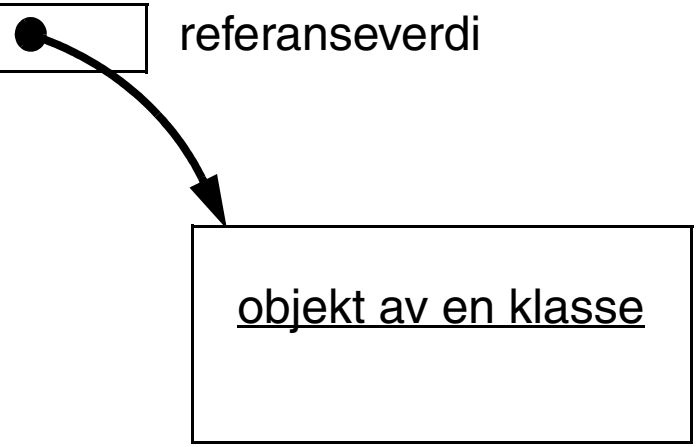
Verdier og objekter i Java

Primitive verdier:

`false`	`boolean`
`47`	`byte`
`'a'`	`char`
`22001`	`short`
`1234567`	`int`
`0xffff0000ffff0000`	`long`
`3.14e1f`	`float`
`3.1415926535897932384`	`double`

Referanser:

`null`	null-referanse
	referanseverdi

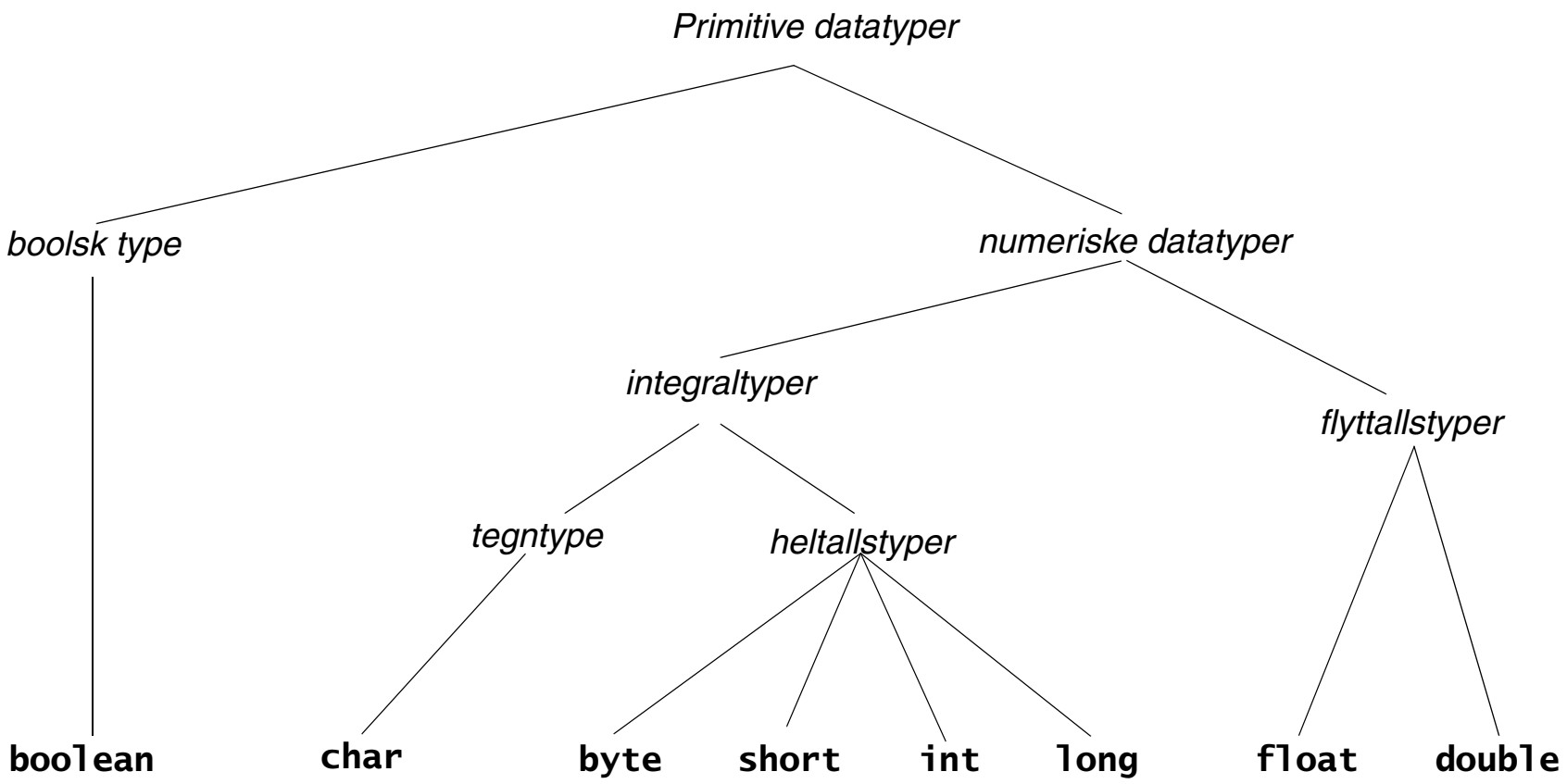


Primitive datatyper og variabler

- *Variabler* brukes til å lagre *verdier*.
- En variabel har *navn*, *type*, *størrelse*, *verdi*, og *adresse*.

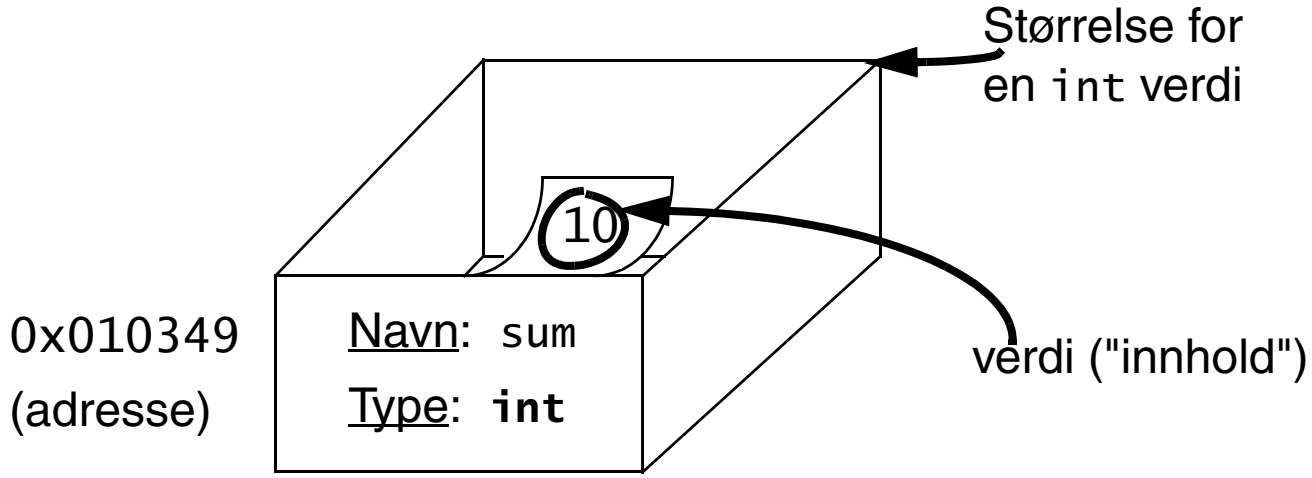
Grunnlegende datatype	Nøkkelord i Java	Verdi-størrelse	Eksempel på *litteral*	Eksempel på *deklarasjon* av en variabel
Heltall: med *fortegn* og *toer-komplement*	`byte`	8-bits	`29`	`byte b;`
	`short`	16-bits	`29 300 -1`	`short m,n;`
	`int`	32-bits	`29 035 0x1D -12345`	`int i,j,k;`
	`long`	64-bits	`29L 1L 123456L`	`long l;`
Flyttall: *IEEE 754-1985 standard*	`float`	32-bits	`1.5F 3.2e2F`	`float r;`
	`double`	64-bits	`1.5 3.2e-2 .1e2 3.14D 0.0 -0.0`	`double s,t;`
Tegn: *UNICODE*	`char`	16-bits	`'a' '\t' '\u0009'`	`char c;`
Boolsk/logisk:	`boolean`	–	**`true false`**	`boolean v;`

Oppsummering: Primitive datatyper



Deklarasjon av variabler

- En *deklarasjon* avsetter lager til variabelen:
  `int sum;`
- Variabelnavn betegner *lokasjon/adresse* i lager der verdien til variabelen lagres.
- Variabelen kan *initialiseres* eksplisitt i en deklarasjon:
  `int sum = 10; // god vane å alltid initialisere variabler!`
- *Verdien* ("innholdet") til variabelen kan manipuleres ved å bruke variabelnavnet.



`int sum = 10;` er ekvivalent med

```
int sum;
sum = 10;
```

Lager og verdier

- En *skriveoperasjon* overskriver gammel verdi i lageret med ny verdi, f.eks. ved tilordning.
- En *leseoperasjon* endrer ikke verdi i lageret, f.eks. en variabel (*operand*) i et uttrykk.

```
int i = 10,
    j = 20;
i = 2 * j;  // leseoperasjon: henting av verdi til j
            // skriveoperasjon: i får verdi 40
```

Uttrykk

(Verdi) type	Uttrykk (har en verdi) av typen
Type	• variabel, litteral • metodekall • typespesifikke uttrykk

`int`	`antall, 2, 97`
	`Math.power(2,10), db.antallBiler()`
	`8+antall, (2*4)/3, 13%2`

`double`	`pris, moms, 3.14D, -0.5e5, .035, 2.667`
	`Math.cos(0.0), Math.sqrt(3.14+1.0)`
	`8*pris+moms, (2.0*4)/3`

Uttrykk (forts.)

(Verdi) type	Uttrykk (har en verdi) av typen
Type	• variabel, konstant eller litteral • metodekall • typespesifikke uttrykk

`boolean`	`ferdig, OK, true, false`
	`Character.isDigit(c), Character.isLetter(c), Character.isLowercase(c)`
	Sammenligningsoperatorer: `antall == 3, i < 100`
	Boolske eller logiske operatorer: `ferdig && OK, (i<100)||(antall == 1)`

`char`	`tegn, 'z', '\n', '\u000a'` `Character.toLowercase(c)`

`String`	`str, "en streng\n", "\"aha\"", ""`
	`str.substring(start, stopp+1)`
	Konkatenering: `"Hei" + " på " + "deg.\n"`

Entydighetsregler: *presedens* og *assosiativitet* til operatorer

- brukes for entydig evaluering av uttrykk.
- reglene kan overstyres ved å bruke parenteser, `()`.

Problem: Er `2 + 3 * 4` lik `20` eller `14`?

- *Presedensregler* brukes til å avgjøre hvilken operator som skal utføres først, dersom det finnes 2 operatorer med *forskjellig presedens* som følger hverandre i uttrykket.
  - Operator med høyest presedens utføres først.

    `2 + 3 * 4` tolkes som `2 + (3 * 4)` (med resultat `14`) siden `*` har høyere presedens enn `+`.

- *Assosiativitetsregler* brukes til å avgjøre hvilken operator som skal utføres først, dersom det finnes 2 operatorer med *samme presedens* som følger hverandre i uttrykket.
  - *Venstre-assosiativitet* medfører *gruppering fra venstre mot høyre*.

    `1 + 2 - 3` tolkes som `((1 + 2) - 3)` siden *binære* operatorer `+` og `-` har venstre-assosiativitet.
  - *Høyre-assosiativitet* medfører *gruppering fra høyre mot venstre*.

    `- - 4` tolkes som `(- (- 4))` (med resultat `4`) siden *unær* operator `-` har høyre-assosiativitet.

Aritmetiske uttrykk

Aritmetisk uttrykk kan bestå av:	**Eksempel:**
– et tall,	`3.14`
– en variabel,	`antall`
– et metodekall,	`Math.sin(90)`
– operator mellom to uttrykk (binær operator),	`fase + Math.sin(90)`
– operator anvendt på et uttrykk (unær operator),	`-rabatt`
– uttrykk i paranteser.	`(3.14-amplitude)`

- Heltallsaritmetikk gir resultat av type `int` med mindre minst en av operandene har type `long`.
- Flyttallsaritmetikk gir resultat av type `double` dersom en av operandene har type `double`.

Aritmetiske uttrykk (forts.)

- Uttrykksberegningen kan medføre *implisitt konvertering* av verdier i uttrykket. *Se flere eksempler nedenfor.*
- Konvertering av "*smalere*" datatyper til "*bredere*" datatyper går (vanligvis) uten tap av informasjon, men omvendt kan det føre til tap av informasjon.
  - Kompilatorer gir melding dersom en operasjon kan føre til en "ulovlig" konvertering.

```
int i = 10;
double d = 3.14;
d = i;       // OK!
i = d;       // Går ikke uten eksplisitt konvertering (cast)
i = (int) d; // Godtatt av kompilatoren.
```

Aritmetiske uttrykk (forts.)

Presedens: Aritmetiske operatorer	
Gruppering	()
Unær	+
	-
Binær	*
	/
(modulo)	%
	+
	-

Uttrykk	Evaluering: Avgjør operander. Evaluer venstre mot høyre.	Verdi
`3 + 2 - 1`	`((3 + 2) - 1)`	4
`2 + 6 * 7`	`(2 + (6 * 7))`	44
`-5+7- -6`	`(((-5)+7)-(-6))`	8
`2+4/5`	`(2+(4/5))`	2
`13 % 5`	`(13 % 5)`	3
`10 / 0`	`ArithmeticException`	
`2+4.0/5`	`(2.0+(4.0/5.0))`	2.8
`4.0 / 0.0`		**`Inf`**

- Unære operatorer *assosierer* fra høyre mot venstre.
- Binære operatorer *assosierer* fra venstre mot høyre.
- *Presedens* avtar nedover i tabellen.
- *Operandene* evalueres fra *venstre mot høyre* før operatoren anvendes.

Heltallsuttrykk:

`3 + 2 -1`	er lik	`4`	
`2 + 6 * 7`	er lik	`44`	
`-5+7- -6`	er lik	`8`	
`2+4/5`	er lik	`2`	*(heltallsdivisjon: trunkering)*
`13 % 5`	er lik	`3`	*(restledd)*

Flyttallsuttrykk:

`2+4.0/5`	er lik	`2.8`	*(flyttallsdivisjon)*
`2.0+4/5`	er lik	`2.0`	*(heltallsdivisjon)*
`4.0 / 0.0`	er lik	`INF`	*(Infinity)*
`0.0 / 0.0`	er lik	`NAN(004)`	*(Not-a-Number)*

Tallgrenser:

```
Integer.MAX_VALUE:   2147483647
Integer.MIN_VALUE:  -2147483648

Long.MAX_VALUE:      9223372036854775807
Long.MIN_VALUE:     -9223372036854775808

Float.MAX_VALUE:     3.40282e+38
Float.MIN_VALUE:     1.4013e-45
```

(Gjelder også for negative flyttall.)

```
Double.MAX_VALUE:    1.79769e+308
Double.MIN_VALUE:    2.22507e-308

Float.POSITIVE_INFINITY:    INF
Float.NEGATIVE_INFINITY:   -INF
```

Tall over-/under-flyt ikke signalisert!

(NB. "Wrap-around" for heltall.)

```
Integer.MAX_VALUE +1:   -2147483648
Integer.MIN_VALUE -1:    2147483647

Long.MAX_VALUE +1:      -9223372036854775808
Long.MIN_VALUE -1:       9223372036854775807

Float.MAX_VALUE +1:      3.40282e+38
Float.MIN_VALUE +1:      1.0

Double.MAX_VALUE +1:     1.79769e+308
Double.MIN_VALUE +1:     1.0

Float.POSITIVE_INFINITY +1:    INF
Float.NEGATIVE_INFINITY -1:   -INF
```

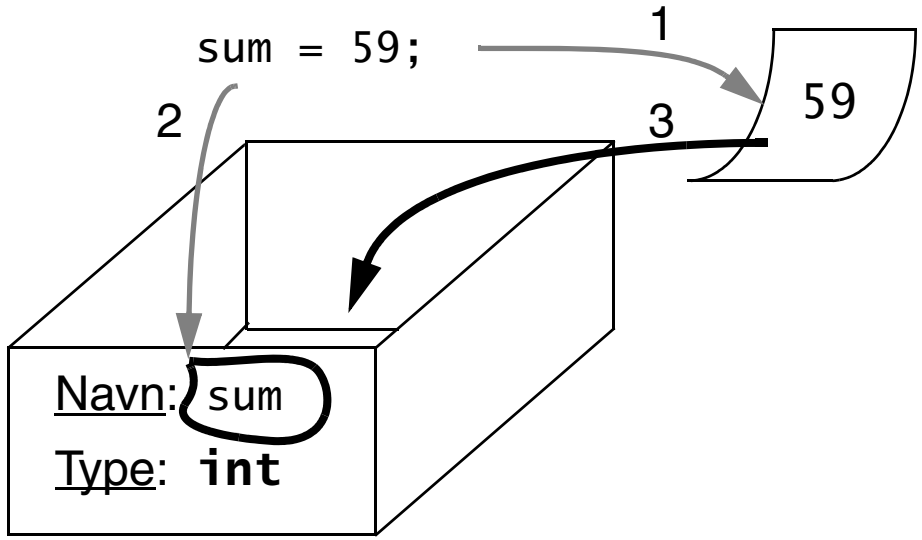
Tilordningsoperator

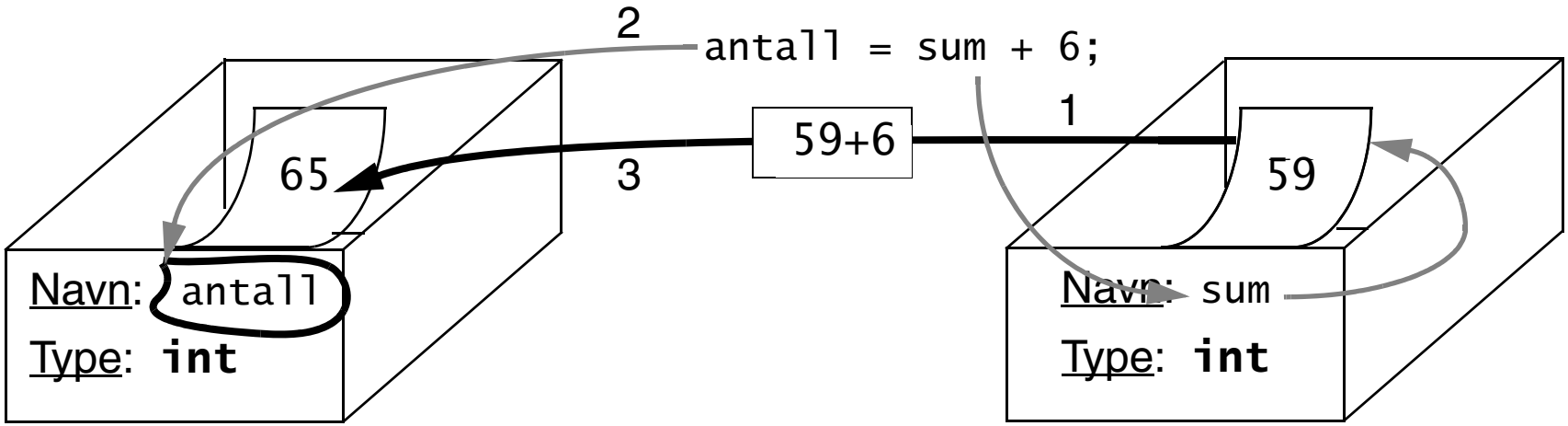
identifikator **=** *uttrykk*

Identifikator ***får*** verdien til *uttrykk*.

Identifikator må være deklarert.

Identifikator og *uttrykk* må være av samme *type*.





Innlesing av fra terminalvinduet

- Det som brukeren skriver på tastaturet blir gjengitt på skjermen.
  - Hvordan kan et program lese det som brukeren skriver på tastaturet?
  - Dette kalles for innlesing fra terminalvinduet.
- Referansen `System.in` angir objektet (*standard inn-enhet*) som tilbyr lesing av *bytes* fra tastaturet.
- Klassen `java.util.Scanner` tilbyr metoder for å konvertere bytes til primitive verdier.
- Saksgangen for å lese verdier fra tastaturet:

```
import java.util.Scanner;                    // Importer Scanner-klassen.
...
Scanner tastatur = new Scanner(System.in);  // Vi må koble en Scanner
                                            // til standard inn-enhet.
// Vi kan nå kalle diverse metoder i Scanner-klassen for å lese verdier
// fra tastaturet.
int i = tastatur.nextInt();       // Les et heltall.
double d = tastatur.nextDouble(); // Les et flyttall.
String str = tastatur.nextLine(); // Les (resten av) linjen som streng.
...
```

- Eksempel:

```
import java.util.Scanner;
/* Celsius til Fahrenheit:
 *    9 x celsius
 *    ----------- + 32
 *         5
 */
public class Temperatur {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.println("Oppgi temperatur i Celsius:");
    double celsius = tastatur.nextDouble();
    double fahr = (9.0 * celsius)/5.0 + 32.0;
    System.out.println(celsius + " Celsius tilsvarer " +
                       fahr + " Fahrenheit.");
  }
}
```

Kjøring av programmet:

```
Oppgi temperatur i Celsius:
23,5
23.5 Celsius tilsvarer 74.3 Fahrenheit.
```

- Merk:
  - Brukeren taster flyttall med komma-notasjon pga norske *locale*.
  - Metodene `print()` og `println()` skriver flyttall med punktum-notasjon.

  Se også `TemperaturII.java`.

Formatert utskrift

- Vi kan kalle metoden `printf()` i `System.out`-objektet (*standard ut-enhet*) for å skrive en *formatert streng* til terminalvinduet, basert på en *formatstreng* og en *argument-liste*:

```
printf(String formatStreng, Object... argumentliste)
```

- En formatstreng kan inneholde både *fast tekst* og *formatspesifikasjoner*.
  - Når man kaller `printf()`-metoden, vil alle argumentene i argumentlisten bli sendt til metoden og formatert i henhold til formatspesifikasjoner i formatstrengen.

Eksempel: formatert utskrift

```
System.out.printf("Formatert utskrift|%6d|%8.3f|kr. |%.2f|%6s%n",
                                  2004, Math.PI, 1234.0354, "hei");
```

Utskrift (norsk tilpasning):

```
Formatert utskrift|  2004|   3,142|kr. |1234,04|  hei!
```

– Formatstrengen er den første parameteren i metodekallet.
– Argumentlisten består av argumenter som skal formatteres ifølge formatstrengen.
  - I dette tilfellet, er det 4 parametere: `2004, Math.PI, 1234.0354, "hei!"`.
– Formatstrengen inneholder 5 formatspesifikasjoner: `%6d`, `%8.3f`, `%.2f`, `%6s` og `%n`.
  - Formatspesifikasjoner spesifiserer *hvordan* argumentene skal prosesseres, og deres *rekkefølge* forteller *hvor* resultat av hver argument skal innsettes i formatstrengen.

Type	Formatspesifikasjon	Argument	Resultat
Heltall	`%6d`	`2004`	`"  2004"`
Flyttall	`%8.3f`	`Math.PI`	`"   3,142"`
Flyttall	`%.2f`	`1234.0354`	`"1234,04"`
Streng	`%5s`	`"hei!"`	`"  hei!"`

– Formatspesifikasjon `%n` gir linjeskift der den forekommer i formatstrengen.

Enkelt valg: Endring av kontrollflyt

Problem: Beregn total ukelønn til en ansatt som har fast ukelønn på kr. 5000, men får ekstra betaling dersom vedkommende har jobbet overtid. Overtidsbetalingen er basert på fast timelønn (kr 200) og antall timer jobbet utover 37.5 timer i uken.

- For en som ikke har jobbet overtid, er ukelønn gitt ved:
  `5000.0`
- For en som har jobbet overtid, er ukelønn gitt ved:
  `5000.0 + (antallTimer - 37.5) * 200.0`

Algoritme for beregning av ukelønn:

```
ukelønn = 5000.0
```

Hvis betingelsen for overtid er sann

```
   ukelønn = ukelønn + (antallTimer - 37.5) * 200.0
```

- Må se på hvordan vi uttrykker *betingelser* og hvordan vi fortar *valg* i Java.

Sannhetsuttrykk

- Betingelser uttrykkes ved hjelp av sannhetsuttrykk.
- Et sannhetsuttrykk har primitiv datatype `boolean`, og kan enten ha verdi sann (`true`) eller usann (`false`).
- Et sannhetsuttrykk kan settes sammen vha *sammenligningsoperatorer* og/eller *logiske operatorer*.

```
// boolske variabler er på lik linje med andre variabler
boolean flagg, OK;
flagg = true;
OK = flagg;
```

(Verdi-) Sammenligningsoperatorer

Likhetsoperatorer:	*a og b er aritmetiske uttrykk.*
`a == b`	`a er lik b?`
`a != b`	`a er ikke lik b?`
Relasjonsoperatorer:	*a og b er aritmetiske uttrykk.*
`a < b`	`a er mindre enn b?`
`a <= b`	`a er mindre eller lik b?`
`a > b`	`a er større enn b?`
`a >= b`	`a er større eller lik b?`

- Alle sammenligningsoperatorer er *binære*.
- Rangering m.h.t. *presedens*:

  1. *Aritmetiske operatorer* 2. *Sammenligningsoperatorer* 3. *Tilordningsoperatorer*

```
int i = 1999;
boolean erPartall = i%2 == 0;          // false
float antallTimer = 56.5;
boolean overtid = antallTimer > 37.5;  // true
```

Kontrollflyt: `if`-setning

- `if`-setning kan brukes til å avgjøre om en handling skal utføres, dvs foreta et valg.

Syntaks:

```
if (<sannhetsuttrykk>) {
    /* setninger for sann del */
}
```

Semantikk:

- Beregn sannhetsuttrykk:
  - hvis sann, utfør setninger for sann del, og fortsett etterpå med resten av programmet.
  - hvis usann, hopp over den sanne delen, og fortsett med resten av programmet.

Eksempel:

```
ukelønn = 5000.0;
if (antallTimer > 37.5) {
    ukelønn = ukelønn + (antallTimer - 37.5) * 200.0;
}
```

Eksempel: `if`-setning

```
import java.util.Scanner;
class Utbetaling1 {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);

    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    double antallTimer = tastatur.nextDouble();
    double ukelønn = 5000.0;                                    // (1)
    if (antallTimer > 37.5) {                                   // (2)
      ukelønn = ukelønn + (antallTimer - 37.5) * 200.0;         // (3)
    }
    System.out.printf("Ukelønnen for %.1f timer er %.2f kroner%n",
                      antallTimer, ukelønn);                    // (4)
  }
}
```

Kjøring av program:

```
Gi antall timer arbeidet [flyttall]: 46,5
Ukelønnen for 46,5 timer er 6800,00 kroner
```

- Merk:
  - Brukeren taster flyttall med komma-notasjon pga norske *locale*.
  - Metoden `printf()` skriver flyttall med komma-notasjon pga norske *locale*.

Setninger

- En setning i Java kan være *enkel* (bestå av én handling) eller *sammensatt* (bestå av flere handlinger gruppert som en *blokk*).

"Lokal" blokk

- Blokk har følgende notasjon: `{<-- begynner en blokk ... avslutter en blokk -->}`
- En blokk brukes til å gruppere en *sekvens av setninger*, og kan ha variabeldeklarasjoner.
- En blokk (med setninger) betraktes som én setning (kalt *sammensatt setning*), og kan brukes der en enkel setning kan brukes.
- Dersom en blokk kun inneholder én setning, kan blokknotasjonen sløyfes.

Avslutningstegn: ;

- Semikolon (`;`) brukes til å *avslutte* en setning.
- En sammensatt setning trenger *ikke* et semikolon for å avslutte den.
- Et semikolon alene, uten en ekte setning, betegner den *tomme setningen* som gjør ingenting.

Mer om valg

Problem: Beregn ukelønn basert på timelønn og antall timer jobbet i uken.

Anta at det betales 50% tillegg for alle timer over `37.5` timer i uken.

1. For en som jobber overtid, er ukelønn gitt ved:
   `timelønn * 37.5 + 1.5 * timelønn * (antallTimer - 37.5)`
2. For en som ikke jobber overtid, er ukelønn gitt ved:
   `timelønn * antallTimer`

Program for beregning av ukelønn må foreta et *valg* om hvilken formel som skal brukes for beregning av ukelønn, avhengig av antall timer jobbet i uken.

Hvis betingelsen for overtid er sann

```
ukelønn = timelønn*37.5 + 1.5*timelønn*(antallTimer-37.5)
```

ellers (dvs at betingelsen er usann)

```
ukelønn = timelønn*antallTimer
```

Kontrollflyt: `if-else`-setning

- `if-else`-setning kan brukes til å velge mellom *to* alternative handlinger.

Syntaks:

```
if (<sannhetsuttrykk>) {
    /* setninger for sann-del */
}
else {
    /* setninger for usann-del */
}
```

Semantikk:

- Beregn sannhetsuttrykk:
  - hvis sann, utfør setninger for sann del, og fortsett etterpå med resten av programmet.
  - hvis usann, utfør setninger for usann del, og fortsett etterpå med resten av programmet.

Eksempel: `if-else`-setning

```
import java.util.Scanner;
class Utbetaling2 {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    double ukelønn = 0.0;
    double antallTimer = tastatur.nextDouble();
    if (antallTimer <= 37.5) {                                  // (1)
      ukelønn = 5000.0;                                         // (2)
    } else {                                                    // (3)
      ukelønn = 5000.0 + (antallTimer - 37.5) * 200.0;          // (4)
    }
    System.out.printf("Ukelønnen for %.1f timer er %.2f kroner%n",
                      antallTimer, ukelønn);
  }
}
```

Kjøring 1:

```
Gi antall timer arbeidet [flyttall]: 35,5
Ukelønnen for 35,5 timer er 5000,00 kroner
```

Kjøring 2:

```
Gi antall timer arbeidet [flyttall]: 46,5
Ukelønnen for 46,5 timer er 6800,00 kroner
```

Logiske operatorer

Matematisk uttrykk:

x < z < y

betyr at *z* er større enn *x* og mindre enn *y*.

Dette er ekvivalent med følgende relasjon:

x < z *og* z< y

Hvordan kan vi teste sannheten av dette uttrykket?

- BRUK LOGISKE OPERATORER!

Kortsluttende *logiske* operatorer

Minkende presedens		*x og y er sannhetsuttrykk*
NEGASJON (NOT)	`!x`	resultatet er motsatt sannhetsverdi til x
OG (AND)	`x && y`	sann dersom begge uttrykkene x og y er sanne
ELLER (OR)	`x || y`	sann dersom minst ett av uttrykkene er sant

- Binære operatorer ELLER (||) og OG (&&) assosierer fra *venstre mot høyre*.
- Unær operator NEGASJON (!) assosierer fra *høyre mot venstre*.

Eksempel: *Kortsluttet* beregning av logiske operatorer

```
boolean b1 = (4 == 2) && (1 < 4) // false - parenteser utelates.
boolean b2 = (!b1) || (2.5 > 8)  // true - parenteser utelates.
boolean b3 = !(b1 && b2)         // true
boolean b4 = b1 || !b3 && b2     // false
```

Beregningsrekkefølge:

```
(b1 || ((!b3) && b2)) => (false || ((false) && b2))
=> (false || (false)) => (false)
```

Nøstede `if`-setninger

Anta følgende pseudokode:

Hvis *bensintanken er mindre enn 3/4 full:*

Kontroller om den er mindre enn 1/4 full og gi melding "Lav på bensin!"

ellers:

Gi melding "Minst 3/4 tank. Tut og kjør!"

Programkode (med logisk feil):

```
if (bensin_tank < 0.75)          // 1.if-setning
    if (bensin_tank < 0.25)      // 2.if-setning
        System.out.println("Lav på bensin!");
else
    System.out.println("Minst 3/4 tank. Tut og kjør!");
```

Hvilken `if`-setning er `else`-del knyttet til?

Svar: `else`-del er *alltid* tilknyttet *nærmeste* `if`-setning.

- Programkoden ovenfor har en *logisk feil*!

Nøstede `if`-setninger (forts.)

Riktig programkode:

```
if (bensin_tank < 0.75) {   // Nødvendig med blokknotasjon
    if (bensin_tank < 0.25) // if-setning i sanndel
        System.out.println("Lav på bensin!");
} else
    System.out.println("Minst 3/4 tank. Tut og kjør!");
```

Nøstede `if`-setninger (forts.)

Hva er galt?

```
if (bensin_tank < 0.75 && bensin_tank < 0.25) {
    System.out.println("Lav på bensin!");
} else {
    System.out.println("Minst 3/4 tank. Tut og kjør!");
}
```

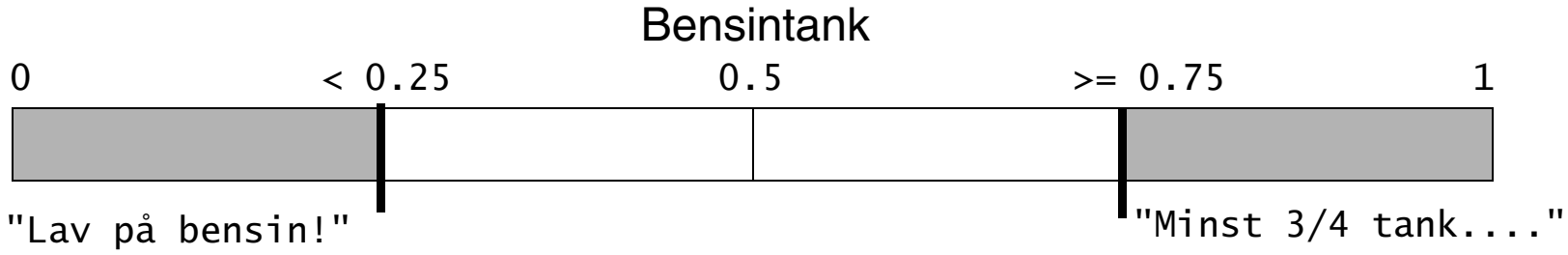
Den 2. meldingen kan bli skrevet selv om tanken *ikke* er 3/4 full!

- Nøstede betingelser kan ikke uten videre sammensettes.

`if (b1)`
`  if (b2)`
`  if (bn)`
`  ...`
`else`
`  ...`
	er ikke nødvendigvis lik
	(likhet dersom uten `else`-del)
		`if (b1 && b2 ... && bn)`
		`  ...`
		`else`
		`  ...`

De Morgans lover

`b1` *og* `b2` *er sannhetsuttrykk*
`!(b1 && b2) <=> (!b1 || !b2)`
`!(b1 || b2) <=> (!b1 && !b2)`



`!(bensin_tank < 0.75 && bensin_tank < 0.25) <=>` `!(bensin_tank < 0.75) || !(bensin_tank < 0.25)`
`!(bensin_tank < 0.75) <=> (bensin_tank >= 0.75)`
`!(bensin_tank < 0.25) <=> (bensin_tank >= 0.25)`
`!(bensin_tank < 0.75 && bensin_tank < 0.25) <=>` `(bensin_tank >= 0.75) || (bensin_tank >= 0.25)`

Nøstede `if`-setninger (forts.)

Bedre løsning:

```
if (bensin_tank < 0.25) {
    System.out.println("Lav på bensin!");
} else if (bensin_tank >= 0.75) { // if-setning i usann del
    System.out.println("Minst 3/4 tank. Tut og kjør!");
}
```

Vanlig feil ved bruk av `if`-setning

```
if (a = b)    // Syntaksfeil: betingelsen ikke sannhetsuttrykk!
    System.out.println("Skjerp deg!");

if (a == b) ; // tom setning
    System.out.println("a er lik b!"); // Logisk feil: alltid utført!

if (a == b) ; // Ingen feil: tom setning betyr "gjør ingenting"
else System.out.println("a er ikke lik b!");
```

Tips: Omslutt sann-del og usann-del i blokk-notasjon, `{}`.

Kaskadevalgsetning

Anta følgende pseudokode:

Skriv "For stort!" dersom tall > svar

Skriv "For lavt!" dersom tall < svar

Skriv "Riktig svar!" dersom tall == svar

Programkode:

```
if (tall > svar) {
    System.out.println("For stort!");
} else if (tall < svar) {
    System.out.println("For lavt!");
} else { // if (tall == svar)
    System.out.println("Riktig svar!");
}
```

Programfeil i Java

Feil under kompilering: *kompileringsfeil*	**Feil under utføring/ kjøring:** *kjørefeil*

• *syntaksfeil*: feil i språkkonstruksjonsoppbygging (*struktur*), f.eks.
`int i = 2 * ; // mangler operand`
• *semantisk feil*: feil i en språkkonstruksjons *mening*, f.eks.
`int i = true; // type inkompatibilitet`
	• utføring av en "ulovlig" operasjon, f.eks.
	`z = x / y; // heltallsdivisjon,`
	`// der y har verdi 0`
	• mangel på ressurser for å kjøre programmet, f.eks.
	- en klasse er ikke funnet
	- data som trengs er ikke tilgjengelig

Feilrapportering: angivelse av sted i kildekode hvor feil har oppstått.	*Feilrapportering*: informasjon om problemspesifikt *unntak*, og utskrift av *metodekall* som er på *programstabelen.* *Mer om dette senere i kurset.*
Feilreparasjon: forsøk på å fortsette kompilering av resten av programmet.	I de fleste tilfeller termineres programmet.

Logiske feil i program

- vanskelig å oppdage.
- krever *testing* av programmet for å avgjøre om det fungerer riktig.
- krever *avlusing* (*debugging*) for å fjerne feil, ved å finne årsaken og rette feilen.
- En enkel måte for avlusing er å bruke `System.out.println()`-metodekall i kildekoden, som skriver ut enkelte resultater beregnet i programmet.

Programlesbarhet

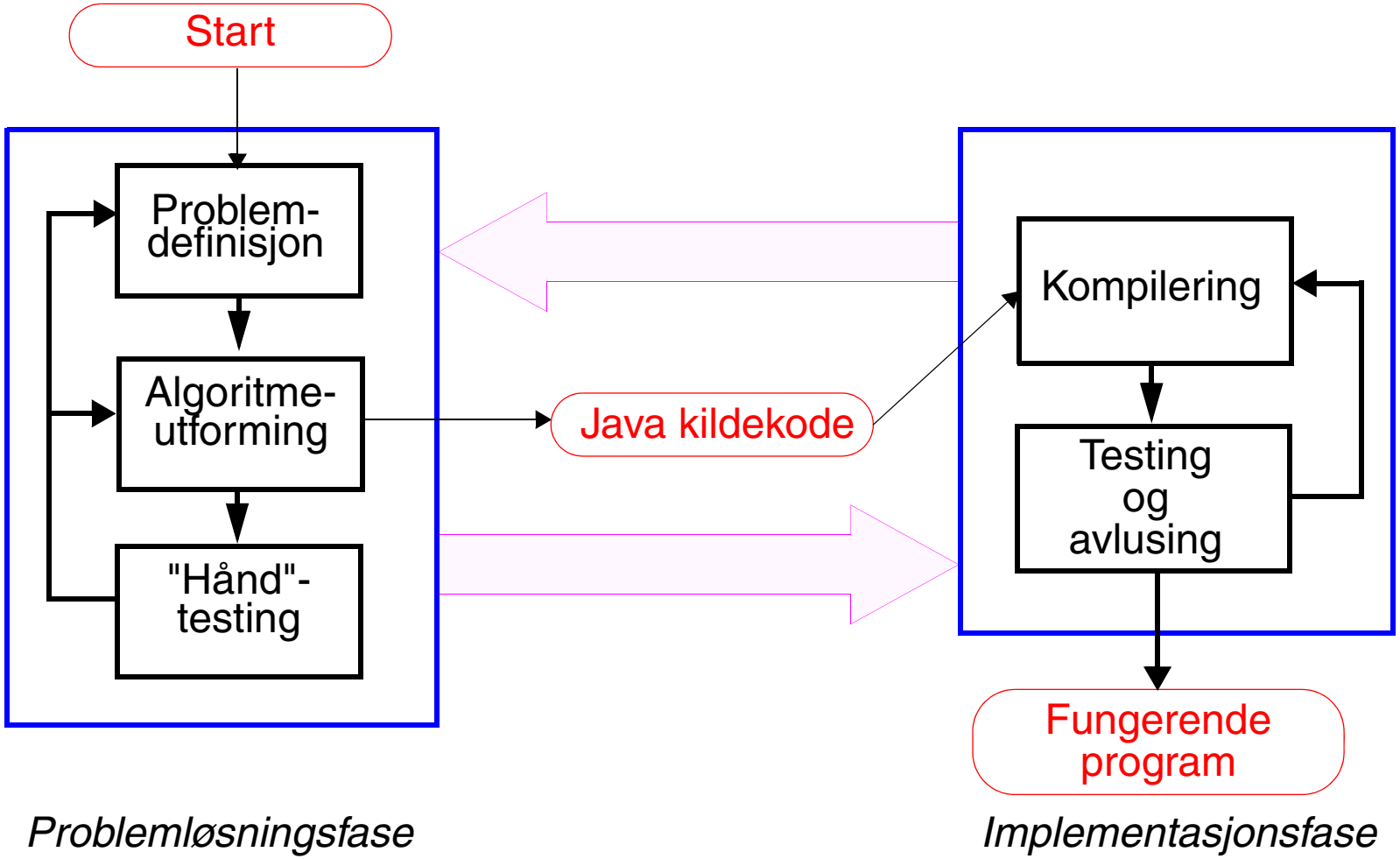
- Programmet er en *sekvens av tegn* som oversettes av kompilatoren.
  – programmet har fritt én-linjes format.
- Bruk *innrykk* for å fremheve språkkonstruksjoner.
- Bruk *fornuftige navn* i programmet.
- Bruk *kommentarer* for å dokumentere programmet.
- Bruk *hvite blanke* (mellomrom, tabulator, linjeskift) for å øke lesbarhet.

Utdrag: Presedensrangering (i avtagende rekkefølge)

Postfiks operatorer	`[] . (`*param-liste*`)` *uttrykk*`++` *uttrykk*`--`
Unære operatorer	`++`*uttrykk* `--`*uttrykk* `+`*uttrykk* `-`*uttrykk* `!`
Opprettelse eller konverteringsoperator	`new` `(`*type*`)` *uttrykk*
Multiplikasjon	`* / %`
Addisjon	`+ -`
Relasjonsoperatorer	`< > >= <= instanceof`
Likhetsoperatorer	`== !=`
Betinget OG	`&&`
Betinget ELLER	`||`
Tilordningsoperatorer	`= += -= *= /= %=`

- *Parenteser*, `()`, kan brukes til å overstyre presedens.
- Alle *binære operatorer* med unntak av tilordningsoperatorer, assosierer fra venstre mot høyre.
- Alle *unære operatorer* assosierer fra høyre mot venstre.

Programutviklingsprosess



Algoritme er en sekvens av presise instruksjoner som fører til en løsning

Programutviklingsprosess

1. Problemdefinisjon og spesifikasjon:
   - prosa som beskriver hva systemet skal modellere.
   - kan skrives av ikke-programmerer.
   - ender opp med *presist* utsagn om hva systemet skal modellere.
   - definerer hvordan bruker skal føre dialog med systemet (*brukergrensesnitt*), andre data som trengs, etc.
2. Utforming (*design*):
   - *splitt og hersk*: systemet består av mindre subsystemer som i sin tur kan bestå av enda mindre subsystemer.
   - OOP: systemet består av en samling av *samarbeidende objekter*
   - *pseudokode* brukes til å beskrive:
     - handlinger eller operasjoner (algoritmeutforming ved *forfining*)
     - relasjoner mellom objekter
   - fremgangsmåten for utforming kan sammenlignes med utforming av hus, broer, kurs, etc.
3. Implementering (*coding*):
   - i Java, med hovedvekt på god *problemoppdeling*, *velstrukturert utforming*, *veldokumentert* programmeringsstil.

Programutviklingsprosess (forts.)

4. Testing og avlusing/feilfinning:
   - *testing*: mate inndata og/eller foreta brukerhandlinger for å se om programmet virker som det skal.
   - *avlusing*: prosess for å finne og fjerne feil i både kode og utforming.
   - *"håndtesting"*: foreta testing uten å bruke datamaskin.

- Resultat:
  - program som fungerer og løser opprinnelig problem
- Krav til program:
  - ferdig innen fristen og sprenger ikke budsjettet
  - tilfredsstiller spesifikasjon eller brukerens forventninger
  - utformet med *bruker* sentralt i betraktningen
  - bruker kreativitet til å løse problem
  - vel-strukturert/-organisert/-skrevet/-dokumentert, kan lett utvides og vedlikeholdes.

Mer om tilordningsoperator

- *Uttrykkssetning*: anvendelse av tilordningsoperator returnerer verdien til *uttrykk* på høyre side av tilordningsoperater som resultat.

```
int a, b, c;
a = b = c = 10; // (a = (b = (c = (10))))
```

- *Utvidete* tilordningsoperatorer: `+=, -=, %=, *=, /=`

<variabel> **op**= *<uttrykk>* betyr *<variabel>* = *<variabel>* **op** (*<uttrykk>*)	**Uttrykkssetning**	**Evaluering**
	`x += a`	`x = x + (a)`
	`x %= a`	`x = x % (a)`
	`x *= a`	`x = x * (a)`

- Utvidet tilordningsoperator `+=` gjelder også for strenger:

```
int a = 2;
String str = "boo";
str += "hoo";   // "boohoo"
str += a; // "boohoo2" - verdi i variabelen a konverteres til streng
a += a;   // a får verdi 4: a = a + a
```

Oversikt: løkker

- Løkker gjør at en del av programmet kan bli utført gjentatte ganger.
  - betingelsen (`true` eller `false`) for å avslutte løkken, kalles for *løkkebetingelse*.
  - den delen av programmet som utføres i en løkke kalles for *løkkens kropp*.
  - som i valgsetninger, kan løkkens kropp være en enkelt handling eller en sammensatt handling (angitt ved en blokk).
- Løkker i Java kommer i 3 varianter:
  1. `while`-løkke
  2. `do-while`-løkke
  3. `for`-løkke
  - forskjellen ligger i forholdet mellom *utføring av løkkekropp* og *test på løkkebetingelse*.
  - Enkelte løkker utfører løkkens kropp før testingen, andre tester før utføringen av løkkens kropp.

`while`-løkke

Syntaks:

```
while (<løkkebetingelse>) {
    /* løkkekropp */
}
```

Semantikk:

- utfør *løkkekroppen* mens *løkkebetingelsen* er **sann**.
  - tester løkkebetingelsen før utføring av løkkekroppen.
  - dersom løkkebetingelsen er usann ved inngangen til løkken, blir ikke løkkekroppen utført i det hele tatt.
- Løkkebetingelsen er et *sannhetsuttrykk*.

```
tall == 0.0
beløp <= saldo
fortsett   // boolsk variabel
(beløp <= saldo) && fortsett
(alder >= 18) && (alder <= 67) || (alder == 80)
```

Inkrement- og dekrementoperatorer

- Variabel-*inkrementoperatorer*:

  `++i` adderer `1` til `i`, bruker deretter `i`: *prefiks*

  `i++` bruker `i`, adderer deretter `1` til `i`: *postfiks*

- Variabel-*dekrementoperatorer*:

  `--i` subtraherer `1` fra `i`, bruker deretter `i`: *prefiks*

  `i--` bruker `i`, subtraherer deretter `1` fra `i`: *postfiks*

Eksempel:

```
// Prefiks orden: inkrement før bruk
int i = 10;
int j = ++i; // j er 11, det er i også.
// Postfiks orden: inkrement etter bruk
int i = 10;
int j = i++; // j er 10, i blir 11.
```

- Nyttige for oppdatering av variabler inne i uttrykk.

  Eksempel: `k = j + ++i; i = --j - -k; i--;`

  `++k` er ekvivalent med `k += 1`

Problemløsing: topp-ned stegvis forfining

Problemstilling: Les flyttall fra terminalen og beregn gjennomsnittsverdien. Innlesning av tall avsluttes ved å inntaste tallverdi `0`.

Grov algoritme:

- *en flyttallsvariabel,* `sum`*, for å lagre summen av tallene.*
- *en heltallsvariabel,* `antall`*, for å lagre antall tall innlest.*

Les tallene;

Skriv ut `(sum/antall)`*;*

Eksempel på dialog:

```
Tast inn tallene. Ett tall pr. linje.
Avslutt med 0,0.
2,0
4,0
3,0
1,0
0,0
Gjennomsnittsverdien er 2,5
```

Algoritmeutforming

Algoritmeforfining I:

1. *les et flyttall*
2. ***hvis** flyttall ikke er 0.0:*

   oppdater sum

   oppdater antall

   les et nytt flyttall

   fortsett med steg 2.
3. *Skriv ut gjennomsnittsverdien.*

- Tallverdien `0.0` brukes som en *vakt* til å avslutte innlesning av tall.

Algoritmeforfining II:

1. *les et flyttall*
2. ***gjenta mens** flyttall ikke er 0.0:*

   oppdater sum

   oppdater antall

   les et nytt flyttall
3. *Skriv ut gjennomsnittsverdien.*

- Steg 2 utgjør en løkke.

- *Topp-ned stegvis-forfining*: handlinger i et *løsningsforslag* deles stegvis i mindre handlinger som fører til mer detaljert løsningsforslag.
- *Pseudokode* brukes til å utforme algoritmer slik at de kan oversettes til et program.

Eksempel: *vakt-kontrollert* repetisjon

```
import java.util.Scanner;
public class Gjennomsnitt {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.println("Tast inn tallene. Ett tall pr. linje.");
    System.out.println("Avslutt med 0,0.");
    int antall = 0;
    double sum = 0.0;
    double tall = tastatur.nextDouble();
    while (tall != 0.0) {
      sum += tall;
      ++antall;
      tall = tastatur.nextDouble();
    }
    if (antall == 0) {
      System.out.println("Du tastet ingen tall!");
    } else { // antall ikke lik 0
      System.out.printf("Gjennomsnittsverdien er %.1f", (sum/antall));
```

```
        }
    }
}
```

Fremgangsmåten ved skriving av et program

1. Problembeskrivelse (Beregn strykprosent)

Anta at 8 studenter tok eksamen i et fag.

Lag et program som

- *for hver student, leser inn om vedkommende fikk bestått (angitt med tall* `1`*) eller fikk stryk (angitt med tall* `2`*).*
- *teller antall bestått og ikke-bestått.*
- *beregner strykprosent (antall som fikk stryk/ antall studenter).*

2. Problemanalyse og presisering

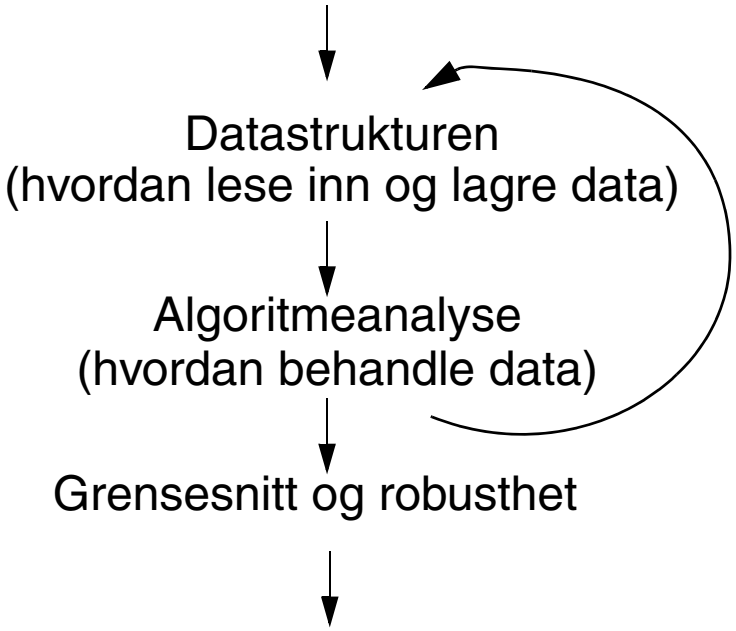
Siden antall studenter er gitt, bruker vi teller-kontrollert løkke.

Vi trenger følgende variabler:

- *en tellervariabel,* `student`*, for å telle hvor mange studenter som er behandlet.*
- *en variabel,* `resultat`*, for å lese inn bestått (*`1`*) eller ikke-bestått (*`2`*).*
- *en tellervariabel,* `bestått`*, for å telle antall bestått.*
- *en tellervariabel,* `ikkeBestått`*, for å telle antall ikke-bestått.*

Fremgangsmåten ved skriving av et program (forts.)

3. Utforming:



4. Koding.

Algoritmeutforming (strykprosent)

Algoritmeforfining I:

1. *Initialiser variabler*
2. ***hvis*** *alle studenter ikke er behandlet:*

   les resultat

   behandle resultat

   oppdater student-teller

   forsett med steg 2.

3. *Beregn strykprosent, etc.*

Algoritmeforfining II:

1. *Initialiser variabler*
2. ***gjenta mens*** *alle studenter ikke er behandlet:*

   les resultat

   hvis *resultat er bestått:*

   øk bestått-teller

   ellers:

   øk ikke-bestått-teller

   oppdater student-teller

3. *Beregn antall bestått/ antall studenter.*

Steg 2 utgjør en løkke.

Eksempel: *teller-kontrollert* repetisjon

```
import java.util.Scanner;
public class Resultat {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    int antallStudenter = 0, bestått = 0, ikkeBestått = 0;
    while (antallStudenter < 8) {
      System.out.print("Tast inn resultat (1=bestått, 2=ikke bestått):");
      int resultat = tastatur.nextInt();
      if (resultat == 1) {
        ++bestått;
      } else {
        ++ikkeBestått;
      }
      ++antallStudenter;
    }
    System.out.println("Antall: " + antallStudenter);
    System.out.println("Antall bestått: " + bestått);
    System.out.println("Antall ikke bestått: " + ikkeBestått);
    System.out.printf("Strykprosent: %.2f%%", (ikkeBestått/antallStudenter)*100.0);
  }
}
```

Utskrift fra programmet:

```
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):2
Tast inn resultat (1=bestått, 2=ikke bestått):3
Tast inn resultat (1=bestått, 2=ikke bestått):43
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):2
Tast inn resultat (1=bestått, 2=ikke bestått):3
Tast inn resultat (1=bestått, 2=ikke bestått):45
Antall bestått: 2
Antall ikke bestått: 6
Strykprosent: 0,00%
```

- Feil i inndata ikke kontrollert, dvs programmet er ikke *robust*!
  - må kontrollere inndata.
- Strykprosent er feil p.g.a. heltallsdivisjon!
  - eksplisitt konvertering v.h.a. konverteringsoperator (cast) nødvendig.
  - konverteringsoperator oppretter en verdi av den angitte typen fra verdien til operanden, og verdien i operanden blir ikke berørt.

```
int i = 20;
double d = (double)i / 16; // d får verdi 1.25D
```

Eksempel: *teller-kontrollert* **repetisjon —** *robusthet*

```
import java.util.Scanner;
public class ResultatII {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    int antallStudenter = 0, bestått = 0, ikkeBestått = 0;
    while (antallStudenter < 8) {
      System.out.print("Tast inn resultat (1=bestått, 2=ikke bestått):");
      int resultat = tastatur.nextInt();
      if (resultat == 1) {
        ++bestått;
        ++antallStudenter;
      } else if (resultat == 2) {
        ++ikkeBestått;
        ++antallStudenter;
      } else {
        System.out.println("Feil i inndata.");
      }
    }
```

```
    System.out.println("Antall: " + antallStudenter);
    System.out.println("Antall bestått: " + bestått);
    System.out.println("Antall ikke bestått: " + ikkeBestått);
    System.out.printf("Strykprosent: %.2f%%",
                      ((double)ikkeBestått/antallStudenter)*100.0);
  }
}
```

Utskrift fra programmet:

```
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):2
Tast inn resultat (1=bestått, 2=ikke bestått):3
Feil i inndata.
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):2
Tast inn resultat (1=bestått, 2=ikke bestått):2
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):1
Tast inn resultat (1=bestått, 2=ikke bestått):1
Antall: 8
Antall bestått: 5
Antall ikke bestått: 3
Strykprosent: 37,50%
```

`do-while`-løkke

Syntaks:

```
do {
    /* løkkekropp */
} while (<løkkebetingelse>);
```

Semantikk:

- utfør *løkkekroppen* mens *løkkebetingelsen* er **sann**.
  - tester løkkebetingelsen etter utføring av løkkekroppen.
  - `do-while`-løkken utfører alltid løkkekroppen minst en gang.

Eksempel: do-while-løkke

```
import java.util.Scanner;
public class ResultatIII {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    int antallStudenter = 0, bestått = 0, ikkeBestått = 0;
    do {
      System.out.print("Tast inn resultat (1=bestått, 2=ikke bestått):");
      int resultat = tastatur.nextInt();
      if (resultat == 1) {
        ++bestått;
        ++antallStudenter;
      } else if (resultat == 2) {
        ++ikkeBestått;
        ++antallStudenter;
      } else {
        System.out.println("Feil i inndata.");
      }
    } while (antallStudenter < 8);
```

```
    System.out.println("Antall: " + antallStudenter);
    System.out.println("Antall bestått: " + bestått);
    System.out.println("Antall ikke bestått: " + ikkeBestått);
    System.out.printf("Strykprosent: %.2f%%",
                      ((double)ikkeBestått/antallStudenter)*100.0);
  }
}
```

Sammenligning: `while` og `do-while`

- I begge løkker:
  - stopper utføring av løkkekroppen dersom løkkebetingelsen blir usann.
  - løkkebetingelsen må ha en lovlig verdi før den testes.
  - sørg for at løkkebetingelsen blir påvirket av en handling i løkkekroppen, slik at den evaluerer til sann, ellers risikerer man en "evig løkke".
  - mest vanlig å bruke *vakt-kontrollerte* løkker, der antall repetisjoner ikke er kjent på forhånd.
- `do-while`-løkke
  - løkkekroppen utført minst én gang.
  - løkkebetingelsen testet ved slutten av løkken, etter løkkekroppen.
- `while`-løkke
  - mulig at løkkekroppen ikke blir utført i det hele tatt.
  - løkkebetingelsen testet ved starten av løkken, før løkkekroppen.
  - løkkevariabler må initialiseres før inngang til løkken.
  - nyttig for å kontrollere feil i data som kan forårsake problemer i løkken, f.eks. `while (objRef != null) { ... objRef.melding() ... }`

Sammenligning: `while` og `do-while` (forts.)

- Hva er forskjellen mellom disse 2 løkkene?

```
...
while (kattenErBorte) {
    mus.leker();
}
...
```

```
...
do {
    mus.leker();
} while (kattenErBorte);
...
```

Påstander

Hvordan kan vi garantere at programmet gjør det som det skal gjøre?

- Dette krever grundig *testing*.

- En måte å bygge tillit til programmet på er å bruke påstander.
- En *påstand* er et utsagn som vi hevder er sant under kjøring av programmet.
  - Utsagnet dreier seg om resultater som blir beregnet under kjøringen.
- Slike påstander kan plasseres i kildekoden, og disse blir utført under kjøring av programmet.
  - Dersom en påstand holder, kan vi gå ut fra at den delen av programmet påstanden gjelder, fungerer riktig.

Påstander: `assert`-setning

Syntaks:

```
assert <sannhetsuttrykk>;                  // den enkle formen
assert <sannhetsuttrykk> : <uttrykk> ;     // den andre formen
```

Semantikk for den enkle formen:

Hvis sannhetsverdien til `<sannhetsuttrykk>` *er* `false`*:*

skrives ut en feilmelding på terminalen;

programmet avbrytes;

ellers: // sannhetsverdien av `<sannhetsuttrykk>` *er* `true`

utføringen fortsetter med første setning etter `assert`*-setningen;*

- Den andre formen fører til at verdien til `<uttrykk>` blir skrevet ut sammen med feilmeldingen, dersom `<sannhetsuttrykk>` er usann.

Kjøring med påstander

- Påstander blir bare utført dersom de er *slått på*:

```
>java -ea Hastighet
```

- Programmet kan også kjøres med påstander *slått av* — da blir *ikke* påstander utført:

```
>java Hastighet
```

Eksempel på påstand — den enkle formen

```
import java.util.Scanner;
/**
 *  Påstand -- enkel form
 */
public class Hastighet {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.println("Oppgi avstand (km):");
    double avstand = tastatur.nextDouble();
    System.out.println("Oppgi tid (timer):");
    double tid = tastatur.nextDouble();
    double hastighet = beregnHastighet(avstand, tid);
    System.out.printf("Hastighet (km/t): %.1f", hastighet);
  }

  /**
   * Beregner hastighet.
   */
```

```
  static private double beregnHastighet(double avstand, double tid) {
    assert avstand >= 0.0 && tid >0.0;
    double hastighet = avstand / tid;
    assert hastighet >= 0.0;
    return hastighet;
  }
}
```

Kompilering og kjøring av `Hastighet`-klassen:

```
>javac Hastighet.java
>java -ea Hastighet
Oppgi avstand (km):
12,0
Oppgi tid (timer):
3,0
Hastighet (km/t): 4,0
```

```
>java -ea Hastighet
Oppgi avstand (km):
15,0
Oppgi tid (timer):
0,0
Exception in thread "main" java.lang.AssertionError
        at Hastighet.main(Hastighet.java:13)


>java Hastighet
Oppgi avstand (km):
16,0
Oppgi tid (timer):
0,0
Hastighet (km/t): Infinity
```

```
>java Hastighet
Oppgi avstand (km):
18,0
Oppgi tid (timer):
5,0
Hastighet (km/t): 3,6
```

Eksempel på påstand — med tilleggsinformasjon

```
import java.util.Scanner;
/**
 * Påstand -- med tilleggs informasjon
 */
public class HastighetII {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.println("Oppgi avstand (km):");
    double avstand = tastatur.nextDouble();
    System.out.println("Oppgi tid (timer):");
    double tid = tastatur.nextDouble();
    double hastighet = beregnHastighet(avstand, tid);
    System.out.printf("Hastighet (km/t): %.1f", hastighet);
  }

  /**
   * Beregner hastighet.
   */
```

```
  static private double beregnHastighet(double avstand, double tid) {
    assert avstand >= 0.0 && tid > 0.0 : "Enten er avstand negativ eller "
        + "tid er mindre eller lik 0.0";
    double hastighet = avstand / tid;
    assert hastighet >= 0.0 : "Hastigheten er mindre enn 0.0";
    return hastighet;
  }
}
```

- Kompilering og kjøring av `HastighetII`-klassen:

```
>javac HastighetII.java
>java  -ea HastighetII
Oppgi avstand (km):
12,0
Oppgi tid (timer):
-3,0
Exception in thread "main" java.lang.AssertionError: Enten er
avstand negativ eller tid er mindre eller lik 0.0
        at HastighetII.main(HastighetII.java:13)
```

```
>java HastighetII
Oppgi avstand (km):
12,0
Oppgi tid (timer):
-3,0
Hastighet (km/t): -4,0
```

Merknader om påstander

- Ikke bruk påstander til å foreta beregninger i programmet.
  - Disse beregningene blir ikke utført dersom påstander er slått av.

```
assert bil.start(); // Ikke utført dersom påstander er slått av.
```

- Ikke bruk påstander til å kontrollere argumentverdier i en metode dersom andre klienter bruker denne ikke-private metoden.
  - Denne kontrollen blir ikke utført dersom påstander er slått av.
  - Kontroll med påstander vil bare skrive ut standard-feilmeldingen (`AssertionError`) som ikke spesifikt forteller hvilket problem som egentlig har oppstått.

Bruksmåte for påstander

- Påstander kan brukes på følgende måter:
  - som *forkrav* for å kontrollere parametre til `private` metoder.
  - som *postkrav* i metoder for å verifisere at metoden har gjort det som forventes.
  - som *klasseinvarianter*, dvs for å validere *objekt-tilstand*.
  - som *interne invarianter*, f.eks. i `else`-klausul til en `if`-setning eller i `default`-etiketten i `switch`-setninger dersom logikken tilsier at disse *aldri* blir utført.

Flere eksempler i boken viser bruk av påstander.