Kapittel 2

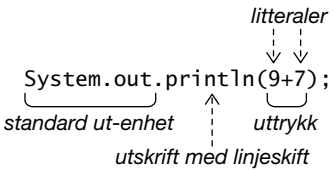
Grunnleggende språkkonstruksjoner

Dette kapitlet beskriver en del grunnleggende språkkonstruksjoner vi må beherske for å kunne skrive programmer. Mange av disse finnes i flere programmeringsspråk, og forståelse av dem er derfor nyttig også dersom man ønsker å lære seg andre språk. Fokus i kapitlet er likevel på hvordan Java realiserer slike grunnleggende konstruksjoner, og på å vise bruken av dem i enkle programmer.

2.1 Utskrift til terminalvindu

Programmer kan regne ut verdier og skrive dem rett ut i terminalvinduet. For eksempel vil metodekallet i Figur 2.1 skrive ut summen av heltallene 9 og 7. Her kalles hvert av tallene 9 og 7 for en *litteral* (eng. *literal*) og 9+7 for et *uttrykk* (eng. *expression*). En litteral er en verdi som står skrevet i kildekoden. Det finnes mange typer uttrykk. Her summerer vi to heltall.

Figur 2.1 *Utskrift av et heltallsuttrykk til terminalvinduet*



Program 2.1 skriver ut strenger og numeriske verdier til terminalvinduet ved (2) til (5) vha. `System.out`-objektet. Dette objektet kalles *standard ut-enhet* (eng. *standard out*), og tilbyr blant annet `println()`-metoden for utskrift *med* linjeskift. Dette objektet tilbyr også en metode ved navn `print`, som *ikke* flytter tekstmarkøren til neste linje etter utskrift. Derfor blir utskriften fra (4) og (5):

```
Verdien av 9 + 7 er 16
```

Vi kompilerer og kjører programmet med følgende kommandoer:

```
> javac EnkelUtskrift.java
> java -ea EnkelUtskrift
```

Kjøremiljøet i Java vil starte utføringen av `main()`-metoden, som må deklareres nøyaktig slik (1) i Program 2.1 viser. Nøkkelordene `public` og `static` spesifiserer at metoden er tilgjengelig utenfra (dvs. fra kjøremiljøet) og tilhører klassen, mens `void` indikerer at metoden utfører

handlinger *uten å returnere noen verdi* til kjøremiljøet. Klammeparantesene, `[]`, angir at parameteren `args` er en tabell. Denne parameteren vil vi senere bruke til å gi verdier inn til et program, men i Program 2.1 er den kun en plassholder for at deklarasjonen skal oppfylle de kravene Java setter til en `main()`-metode.

Program 2.1 *Utskrift av strenger og numeriske verdier til terminalvinduet*

```
class EnkelUtskrift {
  public static void main(String[] args) {                          // (1)
    System.out.println("Verdien av 9 + 7 er ");                     // (2)
    System.out.println(9 + 7);                                      // (3)
    System.out.print("Verdien av 9 + 7 er ");                       // (4)
    System.out.println(9 + 7);                                      // (5)
  }
}
```

Kjøring av programmet:

```
Verdien av 9 + 7 er
16
Verdien av 9 + 7 er 16
```

2.2 Lokale variabler

Programmer kan evaluere uttrykk og skrive resultatet til terminalvinduet umiddelbart. I de fleste tilfeller ønsker vi imidlertid å ta vare på en verdi for å bruke den senere i programmet. Da trenger vi en måte å lagre og hente frem igjen verdier på fra datamaskinens minne.

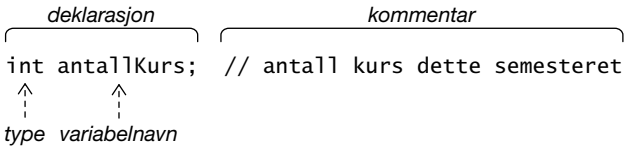
Programmeringsspråk tilbyr variabler for dette formålet. En *variabel* (eng. *variable*) lar programmereren sette av plass i minnet til en verdi og gi denne plassen et *navn*. Dessuten har hver variabel en gitt *datatype* (eng. *datatype*). En datatype (eller bare *type*) er definert ved et sett med gyldige verdier og et sett av operasjoner som kan anvendes på disse variablene.

I dette avsnittet skal vi se på variabler som er definert inne i metoder. Slike variabler kalles *lokale variabler* (eng. *local variables*).

Deklarering av variabler

I Java kan vi deklarere en heltallsvariabel som vist i Figur 2.2.

Figur 2.2 *Variabeldeklarasjon*



Deklarasjonen starter med et *nøkkelord* (eng. *keyword*), `int`, som forteller at variabelen skal

lagre verdier av heltallstype. Deretter følger navnet på variabelen. Tegnet semikolon (;) avslutter deklarasjonen.

Resten av linjen er en kommentar som forteller oss hva variabelen skal brukes til. Bruk av kommentarer i programmer er valgfritt, men det er en god regel å dokumentere variabler dersom formålet med dem ikke er innlysende ut fra navnet.

Hvis vi trenger flere variabler av samme type, kan vi skrive dem i samme deklarasjon, med et komma (,) mellom variabelnavnene, for eksempel slik:

```
int antallKurs, antallStudenter; // antall kurs og studenter i dette semesteret
```

Dette er det samme som å skrive flere deklarasjoner, én per linje:

```
int antallKurs;      // antall kurs dette semesteret
int antallStudenter; // antall studenter dette semesteret
```

Noen eksempler i dette kapitlet utelater imidlertid slike kommentarer, fordi variabelens formål fremgår av navnet, eller linjene er merket med et nummer (*n*) som så diskuteres i teksten.

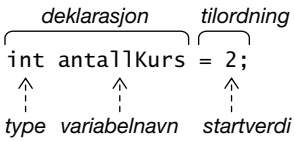
Tilordning av verdi

Etter å ha deklarert hvilken type variabelen skal ha, kan vi *tilordne* den en verdi. Førstegangs tilordning av en verdi til en variabel kalles å *initialisere* (eng. *initialise*) variabelen. Vi kan senere i programmet endre verdien til variabelen ved å bruke en tilordningssetning.

I Java gjøres tilordning ved hjelp av tilordningsoperatoren =. For eksempel kan vi gi en heltallsvariabel verdien 2 med tilordningssetningen

```
antallKurs = 2;
```

Figur 2.3 *Deklarasjon av variabel og tilordning av startverdi*



Program 2.2 inneholder to lokale heltallsvariabler, `antallKurs` og `antallStudenter`. Etter deklarasjon og initialisering skriver vi ut verdien til variabelen `antallKurs` i (1). Vi kan også tilordne en variabel en verdi samtidig med at vi deklarerer den, slik Figur 2.3 og (2) i Program 2.2 viser. Hvis vi senere i programmet vil endre verdien i en variabel, gjøres bare en ny tilordning, slik (3) viser.

En lokal variabel må alltid gis en verdi før vi for eksempel bruker den i en utskriftssetning. Hvis vi prøver å fjerne linjen

```
antallKurs = 2;
```

fra Program 2.2, vil kompilatoren gi en feilmelding. Kompilatoren vil ikke lage noen klassefil, og dermed kan vi ikke kjøre programmet.

Program 2.2 *Et enkelt program med lokale variabler*

```
class LokaleVariabler {
  public static void main(String[] args) {
    int antallKurs;
    antallKurs = 2;
    System.out.print("Antall kurs dette semesteret er ");
    System.out.println(antallKurs);                                    // (1)
    int antallStudenter = 37;                                          // (2)
    System.out.print("Kurset Java-1 har ");
    System.out.print(antallStudenter);
    System.out.println(" studenter");
    antallStudenter = 23;                                              // (3)
    System.out.print("Kurset Java-2 har ");
    System.out.print(antallStudenter);
    System.out.println(" studenter");
  }
}
```

Kjøring av programmet:

```
Antall kurs dette semesteret er 2
Kurset Java-1 har 37 studenter
Kurset Java-2 har 23 studenter
```

Logisk feil

Program 2.3 bruker tre lokale variabler, som tilordnes verdier fra uttrykk. Hver tilordning vil overskrive den forrige verdien. Det er viktig å merke seg at endringer ikke har tilbakevirkende kraft. Dersom vi for eksempel bruker en variabel `a` til å regne ut verdien av en annen variabel `b`, og vi endrer verdien til `a`, må vi passe på at også utregningen blir gjort på nytt. Ellers vil programmet gi tilsynelatende uriktige resultater. Dette skjer i (10) i Program 2.3 der vi, med vitende og vilje, har latt være å regne ut den avhengige variabelen (`grunnflate`) på nytt, etter at vi har endret en av variablene (`bredde`) som inngår i utregningen. Den siste utskriften vil være

```
Grunnflaten av et rektangel med lengde 6 og bredde 5 er: 24
```

som viser at den siste endringen av variabelen `bredde` ikke får tilbakevirkende kraft på variabelen `grunnflate`. Vi har en *logisk feil* i programmet, slik at det ikke oppfører seg på den måten vi ønsker. Siste gang grunnflaten ble beregnet (ved (8)), hadde de to variablene `lengde` og `bredde` verdiene 6 og 4, og dermed blir verdien av grunnflaten 24. For å oppnå det forventede resultat, må grunnflaten beregnes på nytt, ved å gjenta koden i (8) etter (9).

Figur 2.4 viser hvordan verdiene til de lokale variablene endrer seg etter hvert som tilordningene (1) til (9) i Program 2.3 utføres. Farget bakgrunn markerer den verdien som er endret.

Program 2.3 *Tilordning av ny verdi til lokale variabler*

```
class Tilordning {
  public static void main(String[] args) {
    int lengde = 5;                                          // (1)
    int bredde = 4;                                          // (2)
    int grunnflate = lengde * bredde;                        // (3)
    System.out.print("Grunnflaten av et rektangel med lengde ");
    System.out.print(lengde);
    System.out.print(" og bredde ");
    System.out.print(bredde);
    System.out.print(" er: ");
    System.out.println(grunnflate);
    bredde = lengde;                                         // (4)
    grunnflate = lengde * bredde;                            // (5)
    System.out.print("Grunnflaten av et rektangel med lengde ");
    System.out.print(lengde);
    System.out.print(" og bredde ");
    System.out.print(bredde);
    System.out.print(" er: ");
    System.out.println(grunnflate);
    lengde = 6;                                              // (6)
    bredde = 4;                                              // (7)
    grunnflate = lengde * bredde;                            // (8)
    bredde = 5;                                              // (9)
    System.out.print("Grunnflaten av et rektangel med lengde ");
    System.out.print(lengde);
    System.out.print(" og bredde ");
    System.out.print(bredde);
    System.out.print(" er: ");
    System.out.println(grunnflate);                          // (10)
  }
}
```

Kjøring av programmet:

```
Grunnflaten av et rektangel med lengde 5 og bredde 4 er: 20
Grunnflaten av et rektangel med lengde 5 og bredde 5 er: 25
Grunnflaten av et rektangel med lengde 6 og bredde 5 er: 24
```

Figur 2.4 *Tilordning av verdier til variabler (Program 2.3)*

	lengde	bredde	grunnflate
Etter (1)	5		
Etter (2)	5	4	
Etter (3)	5	4	20
Etter (4)	5	5	20
Etter (5)	5	5	25
Etter (6)	6	5	25
Etter (7)	6	4	25
Etter (8)	6	4	24
Etter (9)	6	5	24

Litteraler og konstanter

Litteraler kan brukes ved beregning av heltallsuttrykk, som for eksempel

```
9+7
```

Litteraler brukes også for å beskrive matematiske konstanter (for eksempel *pi*), en faktor det skal multipliseres med i et uttrykk (for eksempel en rentefot), eller en størrelse som angir maksimal kapasitet (for eksempel antall seter i et kinobillettsystem). For å slippe å gjenta litteralen mange steder i koden, er det en fordel å deklarere det faste tallet som en *konstant*.

En konstant er en variabel som ikke kan endre verdi etter at den er initialisert. I Java deklareres en konstant slik:

```
final int ANTALL_DAGER_I_UKEN = 7;
```

Her angir nøkkelordet `final` at variabelen ikke kan endres. Hvis vi prøver å gi den en ny verdi senere i programmet, vil ikke kompilatoren godta dette. Det er vanlig å skrive navn på konstanter med store bokstaver.

Navngivning

Når vi velger navn på variablene våre, er det to ting å huske på: regler for hva som er gyldige navn i Java, og konvensjoner for hva man bør og ikke bør kalle en variabel. Følgende eksempler illustrerer dette:

```
                    // Gyldige og ugyldige variabelnavn
int agent007;       // navn kan inneholde bokstaver og sifre,
//int 1001natt;     // men siffer kan ikke stå først

int mitt_lykketall; // Understrek _ er tillatt
```

```
int antallMinutter, // Java skiller mellom små og store bokstaver
    antallminutter; // så dette er to forskjellige variabler!
                    // Begge navn er gyldige

//int int;          // Nøkkelord som f.eks. int kan IKKE brukes som navn

                    // Konvensjoner for variabelnavn
int antallTimer,    // Variabelnavn har ofte liten bokstav i første ord,
    varePris,       // og stor bokstav i starten av hvert etterfølgende ord
    varePrisMedRabatt;

final int DAGER_I_UKEN = 7,     // Konstanter gis med bare store bokstaver, og
          TIMER_I_UKEN = 7*24;  // understrek _ er vanlig for å skille mellom ord

int nfrgmntr;       // Unngå navn med bare konsontanter

int antallSportsbilerMedFireSylindre; // Ikke bruk mer enn cirka 20 tegn i navnet
```

Nøkkelordene i Java er vist i Tabell B.1 på side 615. Disse kan ikke brukes som variabelnavn.

2.3 Heltall og flyttall

Mange programmeringsspråk tilbyr *primitive datatyper*, slik at man kan definere variabler for å lagre numeriske verdier som inngår i uttrykk. Det er vanlig at ulike typer heltall og flyttall er predefinert som primitive datatyper, der hver type har ulike sett med gyldige verdier, men samme sett av operasjoner. Dette er også tilfelle i Java, som tilbyr i alt seks ulike datatyper for heltall og desimaltall. Vi skal se på de to mest brukte, `int` og `double`, i eksemplene fremover.

Den første typen, `int`, har vi allerede brukt flere ganger. En variabel av denne typen kan lagre et heltall mellom -2^{31} og $+2^{31}-1$. En variabel av type `double` kan lagre flyttallsverdier fra cirka `-1.7E+308` til cirka `+1.7E+308`, med 15 desimalers nøyaktighet.

De andre primitive datatypene i Java blir introdusert utover i boken.

Desimaltegn

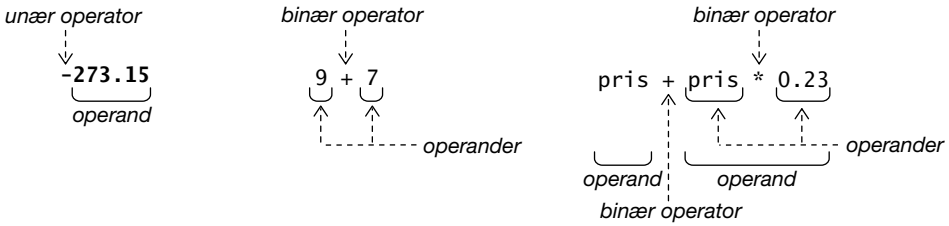
Desimaltegn i Java er `'.'` (punktum). Dermed må alle flyttallslitteraler skrives med punktum som desimaltegn i kildekoden, selv om vi i Norge bruker `','` (komma) som desimaltegn. Vi må f.eks. skrive `3.75` i kildekoden for å angi flyttallsverdien 3,75.

Ved innlesing fra tastatur og utskrift til terminalvinduet er det mulig å sette hvilken *lokal tilpasning* (eng. *locale*) man ønsker å bruke, f.eks. at desimaltegn angis med `','` (komma). Avsnitt 2.6 på side 27 vil vise eksempler på dette.

2.4 Aritmetiske uttrykk og operatorer

De fire vanligste regneoperasjonene på numeriske verdier - multiplikasjon, divisjon, addisjon og subtraksjon - er tilgjengelige i alle vanlige programmeringsspråk. Figur 2.5 illustrerer hvordan noen *aritmetiske uttrykk* er bygd opp ved hjelp av ulike *operatorer* og en eller flere *operander*. En operator angir hvilken regneoperasjon som skal utføres. Hver operator *tar* en eller flere operander som argumenter, det vil si at verdiene til operandene brukes i utregningen for den valgte operasjonen.

Figur 2.5 *Operatorer og operander i aritmetiske uttrykk*



Aritmetiske operatorer og evaluering av uttrykk

Operatorene for multiplikasjon (*), divisjon (/), addisjon (+), subtraksjon (–) og modulo (%), som beregner restleddet ved divisjon, krever alle to operander. Vi sier derfor at de er *binære operatorer*.

Det finnes to *unære operatorer*, + (unær pluss) og – (unær minus), som bare tar én operand. Denne operanden må også være et aritmetisk uttrykk som evaluerer til en numerisk verdi.

Følgende uttrykk er alle gyldige i Java:

`4 + 16` som evalueres til heltallsverdien `20`

`12 / 3` som evalueres til heltallsverdien `4`

`12.0 / 3.0` som evalueres til flytallsverdien `4.0`

`-16.50 / -2.0` som evalueres til flyttallsverdien `8.25`

Med unntak av addisjonsoperatoren + er operatorene ovenfor kun definert for numeriske verdier eller aritmetiske uttrykk.

Uttrykk med strenger og numeriske verdier

Operatoren + kan også brukes til å skjøte sammen to strenger, eller en streng og en numerisk verdi. For eksempel vil kallet

```
System.out.println("Flere strenger kan skrives ut " + "på samme linje");
System.out.println("Det kan også en streng og et tall, f.eks. " + 2);
```

gi utskriften

```
Flere strenger kan skrives ut på samme linje
Det kan også en streng og et tall, f.eks. 2
```

Hver streng må innledes og avsluttes med hermetegn ("). Hvordan operatoren + skjøter sammen to strenger, er forklart i delavsnittet «Strengsammenslåing» på side 62.

Typekonvertering av primitive datatyper

Vi merker oss her at resultatet av uttrykkene ovenfor blir av samme type som operandene. Dersom vi kombinerer numeriske verdier av forskjellig type vha. en binær operator, blir de to verdiene konvertert til en felles type. Dette kalles *typekonvertering*. Noen ganger vil reglene i Java-språket selv sørge for at konverteringen blir utført (*implisitt konvertering*), mens andre ganger må vi selv angi i kildekoden at en bestemt konvertering skal utføres (*eksplisitt konvertering*).

Selv om vi bruker samme operatornavn, for eksempel addisjonsoperatoren + i uttrykk som `4+5` og `12.5+0.3`, vil en aritmetisk binær operator i Java alltid jobbe på to verdier av samme type. Det er altså flere operatorer med navn +, en for addisjon av heltall av type `int`, en for addisjon av flyttall av type `double`, osv. Riktig operator vil bli valgt ut fra typen til operandene i hvert enkelt uttrykk.

Implisitt konvertering

Vi har bare sett på to numeriske datatyper så langt: `int` og `double`. Datatypen `int` kan bare holde heltall, mens datatypen `double` kan holde flyttall og ha et langt større spenn i verdier. Derfor sier vi at `double` er en *bredere datatype* (eng. *broader datatype*) enn `int`. Omvendt sier vi at `int` er en *smalere datatype* (eng. *narrower datatype*) enn `double`.

Dersom vi bruker både heltall og flyttall i et uttrykk, vil Java automatisk konvertere heltallsverdien til en flyttallsverdi før uttrykket evalueres. For eksempel vil heltallsverdien 4 i uttrykket `4 + 12.6` bli konvertert til flyttallsverdien 4,0 før addisjonen utføres. I aritmetiske uttrykk med binære operatorer vil Java automatisk utvide (forfremme) operandverdier til samme type før evaluering utføres.

Tilsvarende vil Java automatisk utføre typekonvertering dersom man tilordner et uttrykk av en gitt type til en variabel som har en bredere type, for eksempel hvis man tilordner verdien av en heltallsvariabel til en flyttallsvariabel, slik som i (1):

```
int antallHeleTimer = 10;
double antallTimer = antallHeleTimer; // (1)
```

Java vil først konvertere heltallet 10 til flyttallsverdien 10,0 (av type `double`) før tilordningen blir utført. En slik konvertering kalles implisitt konvertering og utføres automatisk av kompilatoren.

Eksplisitt konvertering

Dersom vi tilordner en numerisk verdi av en gitt type til en variabel med en smalere type, risikerer vi å miste informasjon. Dette kan for eksempel skje hvis vi vil tilordne verdien av en flyttallsvariabel til en heltallsvariabel. Dersom flyttallsverdien har en desimaldel, kan denne ikke tas vare på i en heltallsvariabel. For å unngå ubevisst informasjonstap krever Java at vi eksplisitt sier at en slik konvertering skal utføres, som i følgende kodebit:

```
double antallTimer = 37.5;
int antallHeleTimer = (int) antallTimer; // (1)
```

Her plasseres den typen det skal konverteres *til* inne i parenteser. Notasjonen (*typenavn*) kalles *eksplisitt typekonvertering* (eng. *casting*). Hvis vi utelater den fra (1) ovenfor, vil ikke kompilatoren godta tilordningssetningen.

Presedens og assosiativitet til operatorer

Evaluering av operander til en aritmetisk operator i Java skjer alltid fra venstre mot høyre. Hvordan de gitte operander grupperes sammen med operatorer bestemmes av operatorenes *presedens*, som angir en innbyrdes ordning mellom ulike operatorer. Dersom to operatorer med *forskjellig presedens* står rett etter hverandre i uttrykket, vil operandene grupperes rundt operatoren med høyest presedens.

For eksempel vil uttrykket `4 + 5 * 2` bli tolket som `4 + (5 * 2)` under kompilering. Dette uttrykket blir evaluert til `4 + 10` og til slutt `14` fordi `*` har høyere presedens enn `+`. Vi har her brukt parenteser til å indikere rekkefølgen kompilatoren har knyttet operander til operatorer på ved tolkningen av uttrykket.

Dersom vi i uttrykket ovenfor ønsker å utføre addisjonen først, må vi eksplisitt angi dette ved hjelp av parenteser

```
(4 + 5) * 2
```

Dette ville gitt resultatet 18.

Tabell 2.1 viser presedensen til de aritmetiske operatorene i Java.

Tabell 2.1 *Presedens til aritmetiske operatorer*

Presedensnivå	Type operator	Operator
høy	unær	`+`, `-`
	binær	`*`, `/`, `%`
lav	binær	`+`, `-`

Assosiativitetsregler brukes til å avgjøre hvilken operator som skal utføres først, dersom det er to operatorer med *samme presedens* rett etter hverandre i uttrykket. *Venstre assosiativitet* med-

fører *gruppering fra venstre mot høyre*, for eksempel vil uttrykket 1 + 2 – 3 tolkes som ((1 + 2) – 3), siden de *binære* operatorene + og – har venstre-assosiativitet. *Høyreassosiativitet* medfører *gruppering fra høyre mot venstre*. For eksempel vil uttrykket – – 4 tolkes som (– (– 4)) (med resultat 4), siden den *unære* operatoren – har høyre-assosiativitet.

Unære operatorer har høyere presedens enn binære operatorer. Multiplikasjon, divisjon og modulo er på samme nivå i presedenshierarkiet og har høyere presedens enn addisjon og subtraksjon. Derfor vil uttrykket `5 * - 6 / 2` bli evaluert slik: `5 * (- 6) / 2`, deretter til `(5 * -6) / 2`, så til `(-30 / 2)` og til slutt `-15`.

Heltalls- og flyttallsdivisjon

De fleste aritmetiske operasjoner i Java gir det resultatet en ville forvente ut fra grunnleggende matematikkunnskaper og erfaring med bruk av en enkel lommekalkulator. Et unntak er en divisjon der både teller og nevner er heltallsuttrykk. *Heltallsdivisjon* gir alltid et heltall som svar. Hvis man for eksempel prøver å regne ut uttrykket `30/4` i et Java-program, vil man få resultatet 7, og ikke 7,5, slik en kalkulator ville gitt. Dersom minst en av operandene til divisjonsoperatoren er et flyttall, utføres en *flyttallsdivisjon*.

Siste del av Program 2.4 viser hva som skjer dersom vi prøver å dividere med null. Merk forskjellen på heltalls- og flyttallsdivisjon. Deler vi et tall `a` på flyttallet 0,0, får vi verdiene *Infinity* og *NaN* (*Not a Number*) avhengig av om `a` selv har verdien 0,0 eller ikke. Heltallsdivisjon med 0 i nevneren fører til feil under kjøring, uansett hva verdien av telleren er. Javas virtuelle maskin skriver ut en feilmelding, og deretter avsluttes programmet. Håndtering av slike feil, kalt unntak, dekkes i kapittel 13.

Merk at ved utskrift av flyttallsverdier ved hjelp av `println()`-metoden brukes `'.'` (punktum) som desimaltegn. Neste avsnitt, "Formatert utskrift" på side 24, viser hvordan vi kan skrive ut flyttallsverdier med `','` (komma) som desimaltegn.

Program 2.4 *Test av divisjons- og modulo-operatoren for heltall og flyttall*

```
class Divisjon {
  public static void main(String args[]) {
    System.out.println("Heltallsdivisjon og modulo:");
    System.out.println(" 3/2 = " + (3/2));
    System.out.println(" 3/4 = " + (3/4));
    System.out.println(" 3%2 = " + (3%2));
    System.out.println("Flyttallsdivisjon og modulo:");
    System.out.println(" 3.0/2.0 = " + (3.0/2.0));
    System.out.println(" 3.0/4.0 = " + (3.0/4.0));
    System.out.println(" 3.0%2.0 = " + (3.0%2.0));
    System.out.println("Divisjon med null:");
    System.out.println(" 2.0/0.0 = " + (2.0/0.0));
    System.out.println(" 0.0/0.0 = " + (0.0/0.0));
    System.out.println(" 2.0/0 = " + (2.0/0));
    System.out.println(" 2/0 = " + (2/0));
  }
}
```

Kjøring av programmet:

```
Heltallsdivisjon og modulo:
 3/2 = 1
 3/4 = 0
 3%2 = 1
Flyttallsdivisjon og modulo:
 3.0/2.0 = 1.5
 3.0/4.0 = 0.75
 3.0%2.0 = 1.0
Divisjon med null:
 2.0/0.0 = Infinity
 0.0/0.0 = NaN
 2.0/0 = Infinity
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Divisjon.main(Divisjon.java:15)
```

Potensielle problemer ved utskrift av strenger og uttrykk i samme kall

Ofte ønsker vi å skrive ut en streng og verdien av et aritmetisk uttrykk i samme metodekall, vha. enten `println()`- eller `print()`-metoden. For eksempel, en forklarende streng etterfulgt av verdien av uttrykket:

```
Verdien av 9 + 7 er 16
```

Da må vi passe på å sette paranteser rundt uttrykket, som i følgende kodebit:

```
System.out.println("Verdien av 9 + 7 er: " + (9 + 7));
```

Ellers vil kompilatoren skjøte operandene i uttrykket sammen som *strenger*, og utskriften blir som følger:

```
Verdien av 9 + 7 er: 97
```

Denne utskriften blir resultatet fra følgende kodebit:

```
System.out.println("Verdien av 9 + 7 er: " + 9 + 7);
```

2.5 Formatert utskrift

Java tilbyr formatert utskrift til terminalvinduet gjennom `printf()`-metoden i `System.out`-objektet:

```
printf(String format, Object... args)
```

Her angir parameteren `format` hvordan formatering skal foregå. Dette er en *formatstreng* (eng. *format string*) der formatspesifikasjoner (eng. *format specifier*s) bestemmer hvordan de etterfølgende parametere skrives ut. Notasjonen «`Object... args`» betyr at metoden kan godta null eller flere parametere.

En formatstreng kan inneholde både fast tekst, dvs. tekst som skrives ut akkurat slik den står, og formatspesifikasjoner, som styrer hvordan de etterfølgende parameterne skrives ut og hvor de skal plasseres i formatstrengen. Når man kaller `printf()`-metoden, vil alle parametere man har gitt i metodekallet bli sendt til metoden som argumenter.

Formateringen av flyttallsverdier er *lokaltilpasset*, dvs. hvilket språk og land utskriften skal følge reglene for. Dette vil for eksempel ha innvirkning på hvilket tegn som brukes som desimaltegn. I Norge vil `','` (komma) bli brukt, mens i blant annet USA vil `'.'` (punktum) bli brukt for å skille mellom helltallsdelen og desimaldelen av et flyttall. I resten av dette kapitlet vil vi stort sett bruke den norske lokaltilpasningen.

Tabell 2.2 viser noen av de mest brukte formatspesifikasjonene i Java. Flere eksempler er gitt i vedlegg I. Alle verdier blir konvertert til strenger ved utskrift med `printf()`-metoden.

Tabell 2.2 *Formatspesifikasjoner i Java*

Parameter	Format-spesifikasjon	Verdi	Utskrift	Kommentar
Heltall	`"%d"`	125	`"125"`	Opptar så mange tegnplasser det er behov for.
	`"%6d"`	125	`"  125"`	Opptar 6 tegnplasser og høyrejusteres. Fylles med mellomromstegn til venstre.
	`"%02d"`	3	`"03"`	Opptar to tegnplasser og fylles med innledende nuller.
Flyttall	`"%f"`	16,746	`"16,746000"`	Opptar så mange tegnplasser som det er behov for, men tar alltid med 6 desimaler.
	`"%.2f"`	16,746	`"16,75"`	Opptar så mange tegnplasser som det er behov for, men tar bare med 2 desimaler.
	`"%8.2f"`	16,746	`"  16,75"`	Opptar 8 tegnplasser (inkludert desimaltegnet) og tar med 2 desimaler.
Strenger	`"%s"`	"Hei!"	`"Hei!"`	Opptar så mange tegnplasser som det er behov for.
	`"%12s"`	"Hei på deg!"	`" Hei på deg!"`	Opptar 12 tegnplasser og høyrejusteres.
	`"%-12s"`	"Hei på deg!"	`"Hei på deg! "`	Opptar 12 tegnplasser og venstrejusteres.
Linjeskift	`"%n"`	(ingen)	(ingen)	Flytter tegnmarkør til neste linje i utskriften.

Utskrift med fast feltbredde

Program 2.5 skriver ut en enkel regning ved hjelp av `printf()`-metoden. På toppen av regningen skrives firmanavnet ut i (1) med en formatstreng som kun inneholder tekst. Dato og klokkeslett skrives så ut på samme linje med to og fire tegns bredde og innledende nuller i (2). En overskrift skrives ut i (3) ved å bruke formatspesifikasjonen `"%-24s"` for kolonnen med varenavn, `"%8s"` for kolonnene med stykkpris og sum, og `"%6s"` for antall. Dette resulterer i en venstrejustert streng med 24 tegnplasser, og høyrejusterte tallkolonner med hhv. 8, 8 og 6 tegnplasser.

Deretter følger utskrift for tre varer i linjene (4)-(5), (6)-(7) og (8)-(9). Disse linjene bruker samme feltbredder som i overskriften, men *konverteringskoder* for flyttall og heltall for de tre siste kolonnene. Stykkpris og sum skrives ut med 8 tegnplasser og 2 desimaler vha. formatspesifikasjonen `"%8.2f"`, og antallet varer skrives ut som et heltall med 6 tegnplasser vha. spesifikasjonen `"%6d"`. Strengene blir venstrejustert, mens alle tallverdier høyrejusteres.

I (10) skrives så totalsummen for alle varene ut med spesfikasjonen `"%8.2f"`. For å få dette tallet til å stå rett under alle varesummene, må vi skrive ut strengen `"Total:"` ved hjelp av spesifikasjonen `"%40s"`. Bredden på 40 tegn finner vi ved å summere bredden av de tre første kolonnene i regningen samt mellomromstegnene mellom dem.

Program 2.5 *Utskrift av en enkel regning*

```
class Regning {
  public static void main(String[] args) {
    double pris, sum, total;
    int antall;
    System.out.printf("Bare Data A/S                 ");                      // (1)
    System.out.printf("%02d-%02d-%04d, %02d:%02d:%02d%n%n",
      6, 7, 2005, 12, 5, 9);                                                   // (2)
    System.out.printf("%-24s %8s %6s %8s%n", "Vare", "Pris", "Antall", "Sum");// (3)
    pris=999.75; antall=2; sum=antall*pris; total=sum;
    System.out.printf("%-24s %8.2f %6d %8.2f%n",                             // (4)
      "Ultraflash, USB 2.0, 1GB", pris, antall, sum);                          // (5)
    pris=1250.00; antall=1; sum=antall*pris; total=total+sum;
    System.out.printf("%-24s %8.2f %6d %8.2f%n",                             // (6)
      "Mega HD, 200GB", pris, antall, sum);                                    // (7)
    pris=63.99; antall=3; sum=antall*pris; total=total+sum;
    System.out.printf("%-24s %8.2f %6d %8.2f%n",                             // (8)
      "USB 2.0 kabel, 2m", pris, antall, sum);                                 // (9)
    System.out.printf("%40s %8.2f%n","Total:", total);                        // (10)
  }
}
```

Kjøring av programmet:

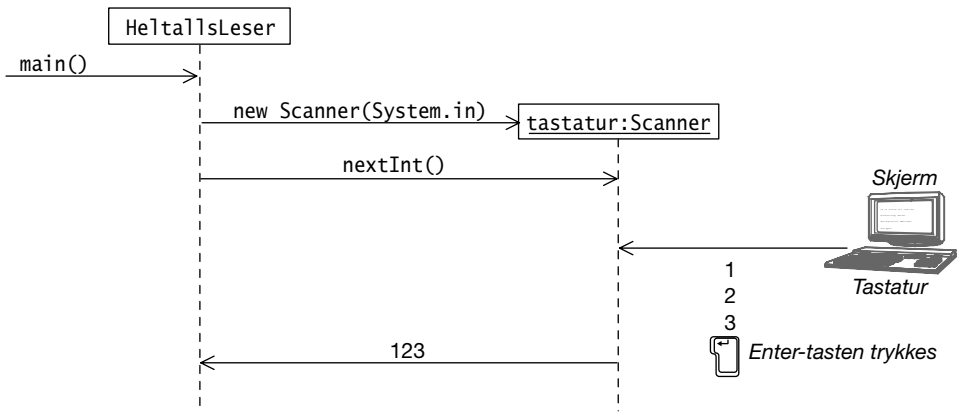
```
Bare Data A/S                 06-07-2005, 12:05:09

Vare                          Pris Antall      Sum
Ultraflash, USB 2.0, 1GB    999,75      2  1999,50
Mega HD, 200GB             1250,00      1  1250,00
USB 2.0 kabel, 2m            63,99      3   191,97
                                   Total:  3441,47
```

2.6 Innlesing av numeriske verdier fra tastaturet

Denne boken bruker en klasse fra Java-biblioteket, `Scanner`, for å lese inn numeriske verdier og strenger fra tastaturet. Denne klassen tilbyr to metoder, `nextInt()` og `nextDouble()`, for innlesing av henholdsvis heltall og flyttall. Kapittel 14 vil forklare hvordan `Scanner`-klassen er implementert, men for å bruke den er det nok å følge stegene angitt nedenfor.

Figur 2.6 viser metodekallene som inngår i innlesingen av et heltall i Program 2.6. Først må vi fortelle kompilatoren at vi ønsker å bruke `Scanner`-klassen. Dette gjør vi ved hjelp av `import`-setningen i (1). Deretter lager (2) et `Scanner`-objekt som knyttes til tastaturet gjennom `System.in`-objektet, som kalles *standard inn-enhet* (eng. *standard in*). Programmet skriver så ut en ledetekst i (3) , slik at brukeren vet hva slags verdi som skal tastes inn. Innlesningen skjer i (4), der den innleste verdien også tilordnes en variabel av riktig type (`int`). I (5) skriver programmet ut igjen det innleste tallet slik at vi kan kontrollere at innlesingen gikk bra.

Figur 2.6 *Innlesing fra tastaturet*



Program 2.6 *Innlesing av ett heltall fra tastaturet*

```
import java.util.Scanner;                                    // (1)
class HeltallsLeser {
  public static void main(String[] args) {
    int innlestTall;
```

```
    Scanner tastatur = new Scanner(System.in);                    // (2)
    System.out.print("Gi et heltall: ");                          // (3)
    innlestTall = tastatur.nextInt();                             // (4)
    System.out.printf("Du ga tallet %d%n", innlestTall);          // (5)
  }
}
```

Kjøring av programmet:

```
Gi et heltall: 123
Du ga tallet 123
```

Innlesing av heltall

Ved hjelp av `Scanner`-klassen kan vi også lese inn verdier for så å bruke dem i beregninger. Program 2.7 viser hvordan vi kan la brukeren oppgi lengde og bredde til et rektangel, og deretter beregne og skrive ut grunnflaten i det.

I (1) og (3) skriver programmet ut ledetekster som forteller brukeren hva slags verdi som forventes. Innlesingen skjer i (2) og (4). I begge tilfeller leses et heltall inn vha. `nextInt()`-metoden. Etter at grunnflaten er beregnet i (5), skrives denne ut i (6). Merk at programmet bruker `print()`-metoden til å skrive ut ledetekster, slik at brukeren kan gi inn en verdi på samme linje.

Program 2.7 *Innlesing av heltallsverdier fra tastaturet*

```
import java.util.Scanner;
class HeltallsAreal {
  public static void main(String[] args) {
    int lengde;
    int bredde;
    int grunnflate;
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi lengden av rektangelet [heltall]: ");                    // (1)
    lengde = tastatur.nextInt();                                                   // (2)
    System.out.print("Gi bredden av rektangelet [heltall]: ");                   // (3)
    bredde = tastatur.nextInt();                                                   // (4)
    grunnflate = lengde * bredde;                                                  // (5)
    System.out.printf( "Grunnflaten av et rektangel med lengde %d og bredde %d er %d%n",
      lengde, bredde, grunnflate);                                                 // (6)
  }
}
```

Kjøring av programmet:

```
Gi lengden av rektangelet [heltall]: 6
Gi bredden av rektangelet [heltall]: 4
Grunnflaten av et rektangel med lengde 6 og bredde 4 er 24
```

Innlesing av flyttall

For å gjøre arealutregningsprogrammet ovenfor mer fleksibelt, kan vi endre det til å støtte innlesing og beregning med flyttall. Med flyttallsvariabler og -verdier, vil kildekoden se ut som i Program 2.8. Alle variabler har nå type `double`, og programmet kaller `nextDouble()`-metoden i (2) og (4), istedenfor `nextInt()`-metoden.

For å hjelpe brukeren har vi også endret ledetekstene i (1) og (3) slik at det går klart frem at flyttall forventes som inndata. Beregninger og utskrift gjøres i (5) og (6), på tilsvarende måte som i Program 2.7. Unntaket er at vi må bruke formatspesifikasjon `"%f"` i (6), siden vi nå skal skrive ut flyttallsverdier i stedet for heltallsverdier.

Program 2.8 *Innlesing av flyttallsverdier fra tastaturet*

```
import java.util.Scanner;
class FlyttallsAreal {
  public static void main(String[] args) {
    double lengde;
    double bredde;
    double grunnflate;
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi lengden av rektangelet [flyttall]: ");          // (1)
    lengde = tastatur.nextDouble();                                       // (2)
    System.out.print("Gi bredden av rektangelet [flyttall]: ");          // (3)
    bredde = tastatur.nextDouble();                                       // (4)
    grunnflate = lengde * bredde;                                         // (5)
    System.out.printf(
      "Grunnflaten av et rektangel med lengde %.2f og bredde %.2f er %.2f%n",
      lengde, bredde, grunnflate);                                        // (6)
  }
}
```

Kjøring av programmet:

```
Gi lengden av rektangelet [flyttall]: 15,5
Gi bredden av rektangelet [flyttall]: 4,25
Grunnflaten av et rektangel med lengde 15,50 og bredde 4,25 er 65,88
```

Mulige feilsituasjoner ved innlesing

Ved innlesing av tall fra tastaturet i Program 2.7 og Program 2.8 er det alltid en mulighet for at brukeren skriver en ugyldig verdi. For eksempel kan han gi en flyttallsverdi under kjøring av Program 2.7, noe som vil føre til at programmet stopper med en feilmelding:

```
Gi lengden av rektangelet [heltall]: 6,5
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at HeltallsAreal.main(HeltallsAreal.java:9)
```

Denne typen feilsituasjon kalles unntak og dekkes i kapittel 13.

`Scanner`-klassen tar hensyn til lokaltilpasning, og vil derfor kreve at flyttallsverdier gis med ',' (komma) som desimaltegn. Dersom bruker gir et flyttall med '.' (punktum) som desimaltegn vil `nextDouble()`-metoden vil skrive en lignende feilmelding som vist ovenfor, og programutføringen avbrytes.

For både heltalls- og flyttallsverdier er det syntaksen til inndata som avgjør om gitt verdi aksepteres eller ikke. `Scanner`-klassen har ingen mekanismer for å fange opp verdier som ikke er fornuftige ut fra det de skal brukes til i programmet. For eksempel kan brukeren gi negative verdier for lengde og bredde i Program 2.7 og Program 2.8 uten at utføringen stopper. Formålsrettet validering er den kallende metoden, i dette tilfellet `main()`, sitt ansvar. `Scanner`-klassen er laget for å kunne brukes i mange sammenhenger, og tilbyr derfor kun syntaktisk validering av inndata.

Innlesing av flere verdier per linje

Ved hjelp av `Scanner`-klassen kan vi lese inn flere verdier fra samme linje på tastaturet, noe Program 2.9 viser. Etter at brukeren har oppgitt to heltall, vil kallet til `nextInt()`-metoden i (1) tilordne den første heltallsverdien til variabelen `tall1`, og kallet til samme metode i (2) tilordne den andre verdien til variabelen `tall2`. Deretter summerer programmet de to gitte tallverdiene og skriver ut resultatet i terminalvinduet ved (3).

Program 2.9 *Innlesing av flere verdier per linje*

```
import java.util.Scanner;
class InnlesingAvFlereVerdier {
  public static void main(String[] args) {
    int tall1,tall2,sum;
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi to heltall: ");
    tall1 = tastatur.nextInt();                                    // (1)
    tall2 = tastatur.nextInt();                                    // (2)
    sum = tall1 + tall2;
    System.out.printf("Summen av %d og %d er %d%n", tall1, tall2, sum); // (3)
  }
}
```

Kjøring av programmet:

```
Gi to heltall: 23 347
Summen av 23 og 347 er 370
```

Tømme resten av linjen under innlesing

Foregående eksempel viste at `Scanner`-klassen kan lese inn *flere* verdier per linje. Dersom vi ønsker at bruker kun skal gi én verdi per linje, kan vi bruke metoden `nextLine()` for å hoppe over resten av inneværende linje.

Program 2.10 kaller denne metoden i (1) for å hindre at bruker oppgir mer enn ett tall ved innlesing av lengden til rektangelet. Selv om bruker gir to tallverdier på første linje, vil kun den første bli brukt i programmet.

Program 2.10 *Tømme resten av linjen ved innlesing*

```
import java.util.Scanner;
class FlyttallsAreal2 {
  public static void main(String[] args) {
    double lengde;
    double bredde;
    double grunnflate;
    Scanner tastatur = new Scanner(System.in);
    System.out.println("Arealberegning - en verdi per linje");
    System.out.print("Gi lengden av rektangelet [flyttall]: ");
    lengde = tastatur.nextDouble();
    tastatur.nextLine();                                          // (1)
    System.out.print("Gi bredden av rektangelet [flyttall]: ");
    bredde = tastatur.nextDouble();
    grunnflate = lengde * bredde;
    System.out.printf(
      "Grunnflaten av et rektangel med lengde %.2f og bredde %.2f er %.2f%n",
      lengde, bredde, grunnflate);
  }
}
```

Kjøring av programmet:

```
Arealberegning - en verdi per linje
Gi lengden av rektangelet [flyttall]: 15,5 3,9
Gi bredden av rektangelet [flyttall]: 4,25
Grunnflaten av et rektangel med lengde 15,50 og bredde 4,25 er 65,88
```

Program 2.8 bruker ikke `nextLine()`-metoden. Derfor kan bruker der gi begge tallene på samme linje, slik denne programkjøringen viser.

```
Gi lengden av rektangelet [flyttall]: 15,5 4,25
Gi bredden av rektangelet [flyttall]: Grunnflaten av et rektangel med lengde 15,50 og
bredde 4,25 er 65,88
```

Programmet vil lese inn den andre flyttallsverdien som gis på første linje, og tilordne denne til variabelen `bredde`. Den andre ledeteksten og resultatet av beregnet grunnflate vil deretter bli skrevet ut på samme linje i terminalvinduet. Programmet gir riktig svar, men utskriften kan virke forvirrende. Hvis man skal lese inn flere tallverdier og ønsker at bruker skal oppgi disse enkeltvis, bør man bruke `nextLine()`-metoden.

2.7 Sannhetsuttrykk

Et sannhetsuttrykk er et uttrykk som har verdien *sant* eller *usant* (eng. *true* eller *false*), gitt ved litteraler `true` og `false`, henholdsvis.

Enkle sannhetsuttrykk

Et *enkelt sannhetsuttrykk* består av én *sammenligningsoperator* og to operander, som kan være variabler, litteraler, konstanter eller aritmetiske uttrykk. For eksempel vil følgende uttrykk i Java

```
antallTimer >= 37.5
```

evalueres til verdien `true` hvis verdien av variabelen `antallTimer` er større eller lik 37,5, og til verdien `false` ellers.

Vi kan også sammenligne en numerisk variabel med en annen eller med en konstant (det vil si at vi sammenligner verdiene deres). Hvis vi har definert en variabel og en konstant, kalt `antallTimer` og `NORMERT_ARBEIDSTID`, vil uttrykket

```
antallTimer >= NORMERT_ARBEIDSTID
```

evalueres til verdien `true` hvis verdien av variabelen `antallTimer` er større eller lik verdien til konstanten `NORMERT_ARBEIDSTID`, og til verdien `false` ellers.

Operandene i sannhetsuttrykk kan være sammensatt av flere uttrykk. Aritmetiske operatorer har høyere presedens enn sammenligningsoperatoren. Et eksempel på dette er

```
antallKrone + antallMynt == antallKast
```

der vi sammenligner summen av to variabler med verdien til en annen. Dette uttrykket tolkes slik: `((antallKrone + antallMynt) == antallKast)`. Her er operatoren `==` testen på likhet, det vil si om begge operandene har samme verdi. Merk at denne operatoren skrives med *to* likhetstegn og uten mellomrom. En vanlig feil er å forveksle den med tilordningsoperatoren `=`.

Tabell 2.3 viser de sammenligningsoperatorene som Java tilbyr.

Tabell 2.3 *Sammenligningsoperatorer i Java*

Operator	Betydning
`==`	er lik
`!=`	er forskjellig fra
`<`	er mindre enn
`>`	er større enn
`<=`	er mindre eller lik
`>=`	er større eller lik

Boolske variabler

Verdien av et sannhetsuttrykk kan tilordnes en *boolsk variabel*. I Java har boolske variabler typen `boolean`, og vi kan deklarere en boolsk (eller logisk) variabel slik:

```
boolean giRabatt = true;
```

Her er verdien `true` en *boolsk litteral*. Boolske variabler i Java kan kun tilordnes verdien `true` eller `false`.

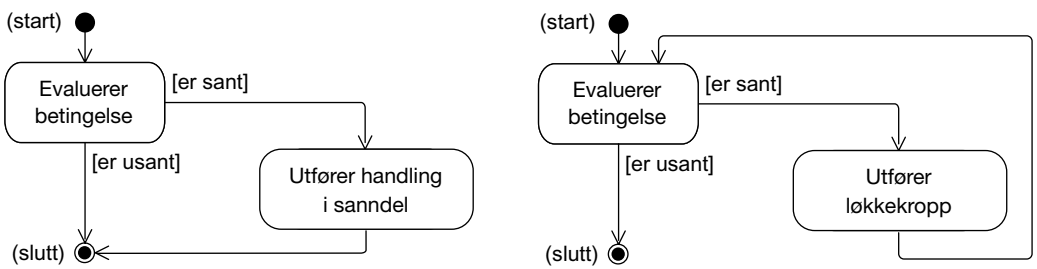
2.8 Kontrollflyt

Ofte ønsker vi å utføre forskjellige handlinger under gitte kriterier. Beregning av lønn er for eksempel ofte avhengig av hvor mange timer man har jobbet. Mange lønnstakere vil få fast lønn så lenge de jobber opp til et normert antall timer per uke, men få ekstra betaling når de jobber overtid. For å løse et slikt problem trenger vi en setning der vi kan stille opp ett eller flere kriterier som må være oppfylt, og i tillegg angi de handlingene som skal utføres i hvert tilfelle. En slik setning kalles en *valgsetning*.

Andre problemstillinger kan kreve at programmet utfører en eller flere handlinger flere ganger. For eksempel vil man i et lønnssystem ha behov for å gjenta beregning av ukelønn for hver enkelt av de ansatte. Denne typen problem kan løses ved hjelp av en *løkke*, der vi kan gjenta en eller flere operasjoner et fast antall ganger eller så lenge et gitt kriterium er oppfylt.

Valg- og repetisjonssetninger gir oss muligheten til å kontrollere hvilke handlinger som skal utføres. Disse setningene bestemmer *kontrollflyt* i programmet, det vil si at de gjør oss i stand til å styre utføringen i programmet, slik at vi ikke lenger er begrenset til å bare utføre sekvensielle handlinger. Figur 2.7 viser kontrollflyten for én form for valgsetning og én form for løkke.

Figur 2.7 *Kontrollflyt i form av valgsetninger og løkker*



(a) Kontrollflyt: valgsetning (b) Kontrollflyt: løkke

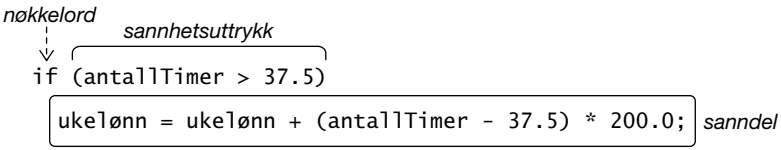
En betingelse for å foreta et valg eller gjenta en handling formuleres som et sannhetsuttrykk. Foreløpig skal vi kun bruke enkle sammenligninger for å styre kontrollflyten. I avsnitt 2.10 på side 40 skal vi se på flere former for sannhetsuttrykk.

2.9 Valgsetninger

Enkel valgsetning

En *enkel valgsetning* utfører en handling hvis en gitt betingelse er oppfylt. Java tilbyr en `if`-setning til dette formålet, der betingelsen er beskrevet som et sannhetsuttrykk, slik Figur 2.8 illustrerer. Hvis sannhetsuttrykket er sant, utføres den betingede handlingen (sanndelen). Hvis uttrykket derimot er usant, fortsetter utføringen rett etter valgsetningen.

Figur 2.8 *Enkel valgsetning*



Program 2.11 bruker en `if`-setning til å beregne ukelønnen for en ansatt. Basis ukelønn for en person som har arbeidet 37,5 timer (eller mindre), er 5000 kroner. Deretter skal det betales 200 kroner for hver etterfølgende time.

I (1) initialiseres variabelen for ukelønn til basislønnen. Deretter evalueres betingelsen for overtid, sannhetsuttrykket `antallTimer > 37.5`, i (2). Dersom dette uttrykket er sant, legges betaling for overtid til basislønnen i (3). Samlet ukelønn skrives så ut i (4).

Program 2.11 *Bruk av enkel valgsetning*

```
import java.util.Scanner;
class Utbetaling1 {
  public static void main(String[] args) {
    double antallTimer; // antall timer arbeidet inneværende uke
    double ukelønn;     // lønn utbetalt per uke
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    antallTimer = tastatur.nextDouble();
    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)
  }
}
```

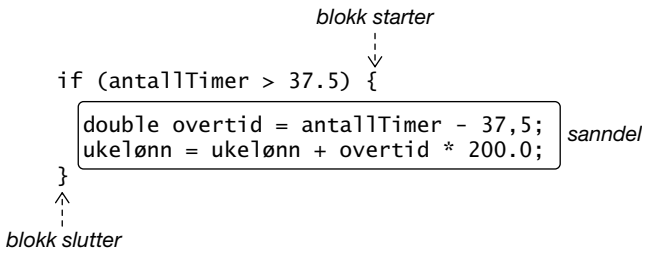
Kjøring av programmet:

```
Gi antall timer arbeidet [flyttall]: 39,5
Ukelønnen for 39,5 timer er 5400,00 kroner
```

Blokker med setninger

I eksemplet ovenfor var det bare en enkelt setning som skulle utføres hvis valgkriteriet var oppfylt (det vil si ble evaluert til `true`). Dersom handlingen hadde bestått av flere setninger, måtte vi ha plassert disse i en *blokk*. En blokk er en sammensatt setning, og kan brukes alle steder der en enkelt setning kan brukes. Starten av en blokk angis med { og slutten av en blokk med }, slik Figur 2.9 viser.

Figur 2.9 *En blokk av setninger*



Variabel definert inne i en blokk

Vi kan definere nye variabler inne i en blokk, slik Figur 2.9 viser. Slike lokale variabler er kun tilgjengelige innenfor blokken. Kompilatoren gir en feilmelding dersom vi forsøker å skrive ut verdien av variabelen overtid ved (1) i Program 2.12.

Program 2.12 *Variabler definert inne i en blokk*

```
import java.util.Scanner;
class Utbetaling1b {
  public static void main(String[] args) {
    double antallTimer; // antall timer arbeidet inneværende uke
    double ukelønn;     // lønn utbetalt per uke
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    antallTimer = tastatur.nextDouble();
    ukelønn = 5000.0;
    if (antallTimer > 37.5) {
      double overtid = antallTimer - 37.5;
      ukelønn = ukelønn + overtid * 200.0;
    }
    System.out.printf("Ukelønnen for %.1f timer er %.2f kroner%n",
      antallTimer, ukelønn);
    //System.out.println("Antall timer overtid: " + overtid);      // (1)
  }
}
```

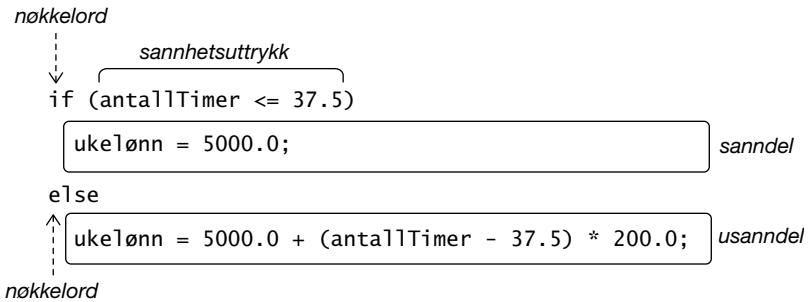
Kjøring av programmet:

```
Gi antall timer arbeidet [flyttall]: 39,5
Ukelønnen for 39,5 timer er 5400,00 kroner
```

Valgsetning med to alternativer

Svært ofte har vi bruk for å velge mellom to alternative handlinger, der programmet utfører det ene alternativet hvis den gitte betingelsen er sann, og det andre alternativet hvis betingelsen er usann. Java tilbyr en `if-else`-setning for dette formålet. Figur 2.10 viser et eksempel på en slik setning.

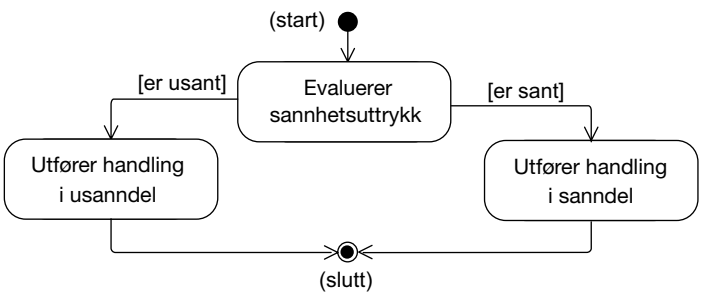
Figur 2.10 *Valgsetning med to alternativer*



Første del av en `if-else`-setning er den samme som for en enkel valgsetning. Sanndelen utføres dersom sannhetsuttrykket er sant, og usanndelen utføres ellers. Nøkkelordet `else` innleder usanndelen.

Figur 2.11 illustrerer utføringen av en `if-else`-setning. Avhengig av sannhetsuttrykket blir bare ett av alternativene utført.

Figur 2.11 *Utføring av en valgsetning med to alternativer*



Program 2.13 løser samme problem som Program 2.11 ved hjelp av en en `if-else`-setning, der den første betingede handlingen beregner basis ukelønn, og den andre beregner lønnen for en ansatt som har arbeidet overtid.

Linje (1) definerer kriteriet for å få utbetalt basislønn, dvs. at den ansatte har jobbet 37,5 timer eller mindre. Dersom dette er oppfylt, settes variabelen ukelønn lik basislønn i (2). Nøkkelordet `else` i (3) markerer starten på den alternative handlingen, mens beregning av ukelønn for en ansatt som har jobbet mer enn 37,5 timer gjøres i linje (4).

Program 2.13 *Valgsetning med to alternativer*

```
import java.util.Scanner;
class Utbetaling2 {
  public static void main(String[] args) {
    double antallTimer; // antall timer arbeidet inneværende uke
    double ukelønn;     // lønn utbetalt per uke
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    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);
  }
}
```

Kjører vi Program 2.13 med samme inndata som i Program 2.11, får vi samme svar:

```
Gi antall timer arbeidet [flyttall]: 39,5
Ukelønnen for 39,5 timer er 5400,00 kroner
```

Prøver vi med litt andre inndata, får vi for eksempel

```
Gi antall timer arbeidet [flyttall]: 42,5
Ukelønnen for 42,5 timer er 6000,00 kroner
```

Nøstede valgsetninger

En handling som utføres i en valgsetning kan være en ny valgsetning. I Program 2.14 har valgsetningen i (1) en valgsetning i (5) som er nøstet i usanndelen. Her kalles (1) for en *nøstet valgsetning*. Ved nøsting av valgsetninger må vi passe på at riktig handling blir utført. Blokknotasjon kan være nødvendig for å angi eksplisitt hvilke handlinger som utføres når en nøstet valgsetning blir utført.

Program 2.14 *Bruk av nøstet valgsetning*

```
import java.util.Scanner;
class Utbetaling3 {
  public static void main(String[] args) {
    double antallTimer; // antall timer arbeidet inneværende uke
    double ukelønn;     // lønn utbetalt per uke
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall timer arbeidet [flyttall]: ");
    antallTimer = tastatur.nextDouble();
    if (antallTimer <= 37.5)                          // (1) nøstet valgsetning
      ukelønn = 5000.0;                               // (2) sanndel
    else {                                            // (3) usanndel-blokk
      ukelønn = 5000.0 + (antallTimer - 37.5) * 200.0; // (4)
      if (antallTimer > 42.0)                         // (5) valgsetning
```

```
        ukelønn = ukelønn + 500.0;                    // (6)
    }                                                 // (7)
    System.out.printf("Ukelønnen for %.1f timer er %.2f kroner%n",
      antallTimer, ukelønn);
  }
}
```

To kjøringer av programmet er:

```
Gi antall timer arbeidet [flyttall]: 39,5
Ukelønnen for 39,5 timer er 5400,00 kroner
```

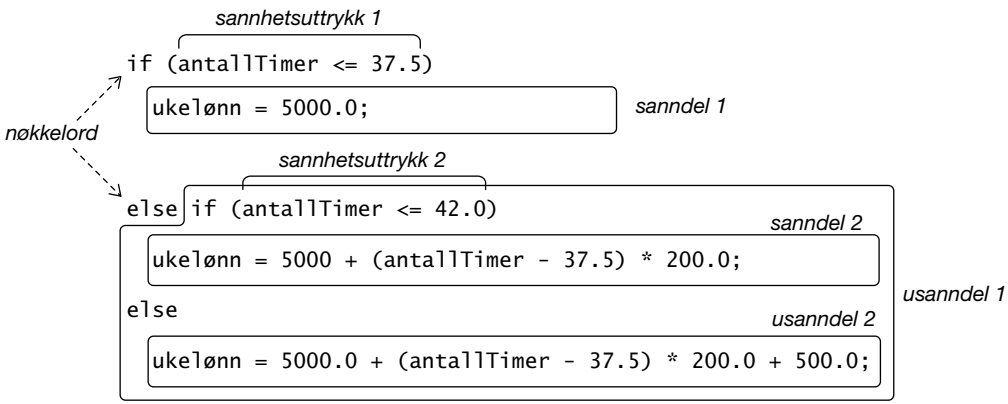
og:

```
Gi antall timer arbeidet [flyttall]: 42,5
Ukelønnen for 42,5 timer er 6500,00 kroner
```

Kaskadevalgsetning

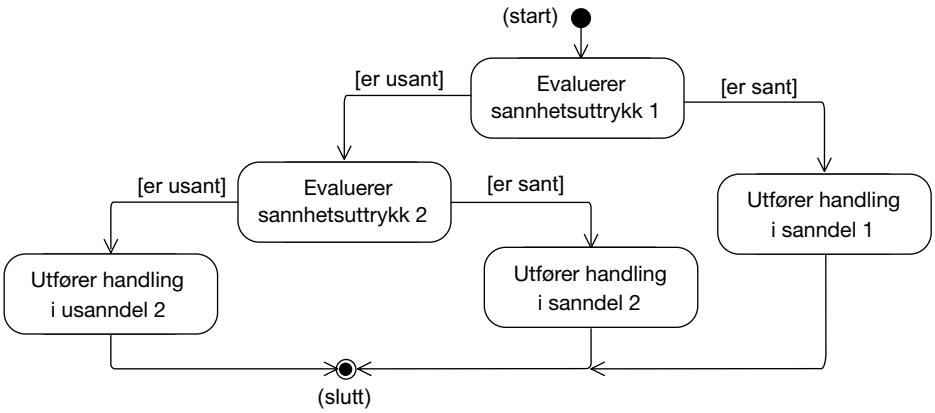
Vi kan også ha en ny `if`-setning som det andre alternativet i en valgsetning, slik Figur 2.12 viser. Her vil den andre `if`-setningen bli utført som én setning dersom det første sannhetsuttrykket, `antallTimer < 37.5`, er usant. I så fall vil sannhetsuttrykk nummer to, `antallTimer < 42.0`, evalueres, og en av handlingene under den andre valgsetningen vil bli utført. Denne varianten av en valgsetning med to alternativer kalles en *kaskadevalgsetning*.

Figur 2.12 *Kaskadevalgsetning*



Figur 2.13 viser utføringen av en kaskadevalgsetning. Ved å la handlingen under *usanndel 2* igjen være en ny valgsetning, kan vi utvide med så mange valg vi ønsker.

Figur 2.13 *Utføring av en kaskadevalgsetning*



Program 2.15 viser bruken av kaskadevalgsetningen fra Figur 2.12 . Dette eksemplet baserer seg på samme problemstilling som de foregående, men nå med det tillegget at personer som har jobbet mer enn 42 timer i en uke, får et ekstratillegg på 500 kroner på toppen av overtidsbetalingen.

Her evalueres først sannhetsuttrykket i (1), `antallTimer < 37.5`. Dersom det er sant, utføres setningen i (2). Hvis det derimot *ikke* er sant, blir sannhetsuttrykket i (3), `antallTimer <= 42.0`, evaluert. Dersom dette uttrykket er sant, utføres setningen i (4). I motsatt fall utføres den setningen som står etter den siste `else`-delen, i (5).

Program 2.15 *Bruk av kaskadevalgsetning*

```
import java.util.Scanner;
class Utbetaling4 {
  public static void main(String[] args) {
    double antallTimer;
    double ukelønn;
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall arbeidstimer [flyttall]: ");
    antallTimer = tastatur.nextDouble();
    if (antallTimer <= 37.5)                                    // (1)
      ukelønn = 5000.0;                                         // (2)
    else if (antallTimer <= 42.0)                               // (3)
      ukelønn = 5000.0 + (antallTimer - 37.5) * 200.0;          // (4)
    else
      ukelønn = 5000.0 + (antallTimer - 37.5) * 200.0 + 500.0;  // (5)
    System.out.printf("Ukelønnen for %.1f timer er %.2f kroner%n",
            antallTimer, ukelønn);
  }
}
```

Test av alternativ 1:

```
Gi antall arbeidstimer [flyttall]: 35,5
```

```
Ukelønnen for 35,5 timer er 5000,00 kroner
```

Test av alternativ 2:

```
Gi antall arbeidstimer [flyttall]: 39,5
Ukelønnen for 39,5 timer er 5400,00 kroner
```

Test av alternativ 3:

```
Gi antall arbeidstimer [flyttall]: 42,5
Ukelønnen for 42,5 timer er 6500,00 kroner
```

2.10 Sammensatte sannhetsuttrykk

Logiske operatorer

Vi kan kombinere flere enkle sannhetsuttrykk ved hjelp av *logiske operatorer* (også kalt *boolske operatorer*). Da får vi et *sammensatt sannhetsuttrykk* som består av to eller flere enkle sannhetsuttrykk kombinert med logiske operatorer. Tabell 2.4 viser de logiske operatorene i Java.

Tabell 2.4 *Logiske operatorer i Java*

Operator	Betydning
`!`	Negasjon, som gir den motsatte boolske verdien av operanden som resultat. Det vil si at `!true` evalueres til `false` og `!false` evalueres til `true`.
`&&`	Betinget OG, som gir resultatet `true` dersom begge operandene har verdien `true`, og resultatet `false` ellers.
`||`	Betinget ELLER, som gir resultatet `true` dersom minst en av operandene har verdien `true`, og resultatet `false` ellers.

Evalueringsregler

De logiske operatorene har *lavere presedens* enn sammenligningsoperatorer, slik at i uttrykket

```
ukedag >= 6 || ukedag == 3
```

vil Java først beregne det enkle sannhetsuttrykket `ukedag >= 6`, deretter det enkle sannhetsuttrykket `ukedag == 3`, og til slutt kombinere verdien av disse med den logiske operatoren `||`. Hvis variabelen `ukedag` inneholder dagnummer i uken nummerert fra 1 for mandager, vil denne testen returnere verdien `true` for ukedagene onsdag, lørdag og søndag, og `false` ellers.

Presedensen til de logiske operatorene er som følger: Negasjonsoperatoren har høyere presedens enn betinget *OG*, som igjen har høyere presedens enn betinget *ELLER*. Med andre ord vil følgende uttrykk med tre boolske variabler `b1`, `b2` og `b3`

```
b1 || !b2 && b3
```

bli tolket som

```
b1 || ((!b2) && b3)
```

Assosiativitetssreglene er som følger: Negasjonsoperatoren assosierer fra høyre mot venstre. Betinget OG og betinget *ELLER* assosierer begge fra venstre mot høyre. Derfor vil variabelen `b2` ovenfor bli assosiert med `!`-operatoren, og uttrykket `!b2` inngå som operand til `&&`-operatoren, som assosieres med to operander fra venstre mot høyre.

I likhet med aritmetiske uttrykk, evalueres sammensatte sannhetsuttrykk i Java fra venstre mot høyre. Derimot kan verdien av slike uttrykk i noen tilfeller bestemmes uten å evaluere alle deluttrykkene, og beregningen vil stoppe så snart verdien av hele uttrykket er bestemt. Vi kaller dette *kortsluttende* eller *forkortet beregning* av sannhetsverdien (eng. *short-circuit evaluation*). For eksempel vil uttrykket

```
(4 == 3) && (3 < 4)
```

evalueres til `false && (3 > 4)`. Deretter vil beregningen av uttrykket stoppe fordi operatoren `&&` (betinget OG) alltid vil gi `false` til svar dersom en av operandene er `false`. Tilsvarende vil uttrykket

```
(4 > 3) || (5 < 4)
```

evalueres til `true || (5 < 4)`. Deretter stoppes beregningen av uttrykket fordi hele uttrykket må være sant når en av operandene til operatoren `||` er `true`.

Program 2.16 viser noen eksempler på sammensatte sannhetsuttrykk som kriterier i valgsetninger. Her må vi i (1) skrive paranteser rundt uttrykket `ukedag == 3` for å kunne utføre negasjonsoperatoren på det. Dette er fordi !-operatoren har høyere presens enn sammenligningsoperatorer.

Program 2.16 *Bruk av diverse sannhetsuttrykk*

```
import java.util.Scanner;
class DiverseSannhetsuttrykk {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi ukedag [1=mandag,...,7=søndag]: ");
    int ukedag = tastatur.nextInt();
    if (ukedag == 2 || ukedag == 4)
      System.out.println("Tirsdag/torsdag: Forelesning i Java fra 12:15-14");
    else if (ukedag==1 || ukedag==5)
      System.out.println("Mandag/Fredag: Java øvelse fra 14:15-16");
    else if (ukedag>=6 || ukedag==3)
      System.out.println("Ingen forelesninger i dag: Repeter det du er usikker på!");
    if (ukedag<=5 && !(ukedag==3))                                          // (1)
      System.out.println("Ukedager unntatt onsdag: Forelesning i logikk fra 10:15-12");
  }
}
```

To kjøringer av programmet er:

```
Gi ukedag [1=mandag,...,7=søndag]: 4
Tirsdag/torsdag: Forelesning i Java fra 12:15-14
Ukedager unntatt onsdag: Forelesning i logikk fra 10:15-12
```

og:

```
Gi ukedag [1=mandag,...,7=søndag]: 6
Ingen forelesninger i dag: Repeter det du er usikker på!
```

De Morgans lover

I noen tilfeller kan kompliserte sannhetsuttrykk forenkles ved hjelp av De Morgans lover:

`!(b1 && b2)` er ekvivalent med `(!b1 || !b2)`

`!(b1 || b2)` er ekvivalent med `(!b1 && !b2)`

når `b1` og `b2` begge er sannhetsuttrykk.

Hvis vi for eksempel vil avgjøre om et heltall `t` ligger innenfor et gitt intervall med nedre grense `a` og øvre grense `b`, kan vi skrive dette som `t >= a && t <= b`, eller som `!(t < a || t > b)`, ved å bruke De Morgans andre lov. Begge disse uttrykkene vil bli evaluert til samme sannhetsverdi for alle verdier av `a`, `b` og `t`, og vi står derfor fritt til å velge den formen vi synes er enklest å forstå.

At et dårlig kjøp er verken billig eller bra, kan formuleres som `(!billig || !bra)` eller som `!(billig && bra)`. Disse to sannhetsuttrykkene er ekvivalente ifølge De Morgans første lov.

2.11 Løkker

En annen nyttig språkkonstruksjon gir muligheten for å gjenta utføring av en handling flere ganger. En slik språkkonstruksjon kalles en *løkke* eller en repetisjonssetning. Handlingen som gjentas i løkken, kalles *løkkekroppen*, og kan bestå av null eller flere setninger.

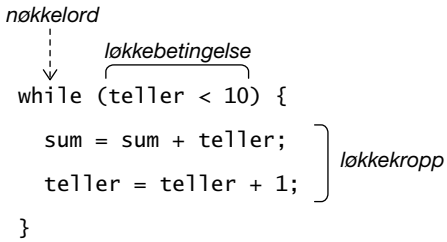
En form for løkke gjentar utføringen av løkkekroppen så lenge en betingelse er oppfylt. Løkkebetingelsen må være et sannhetsuttrykk som evalueres til `true` eller `false`.

Java tilbyr to slike former for løkker: `while` og `do-while`. Forskjellen på disse ligger i når betingelsen for å utføre løkkekroppen skal testes. Hvilken av disse løkketypene man velger vil avhenge av problemet som skal løses. I mange tilfeller kan likeverdige løsninger utformes med begge disse løkketypene, og da blir valg av løkketype mer en personlig preferanse.

Løkke med test før løkkekroppen

I en `while`-løkke i Java kommer testen før løkkekroppen utføres. Figur 2.14 viser en slik løkke. Løkken innledes med nøkkelordet `while` før løkkebetingelsen. Deretter kommer løkkekroppen, i dette tilfellet en blokk, som skal utføres for hvert gjennomløp av løkken.

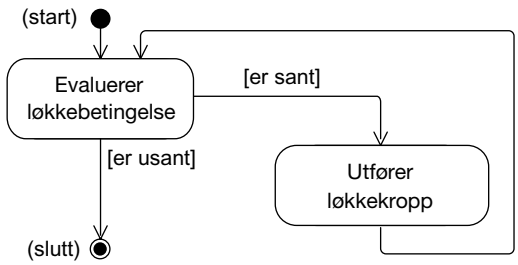
Figur 2.14 *Eksempel på `while`-løkke*



Figur 2.15 viser hvordan en `while`-løkke utføres. Først evalueres løkkebetingelsen. Dersom dette sannhetsuttrykket er sant, utføres løkkekroppen. Etter dette går kontrollen tilbake til løkkebetingelsen som evalueres på nytt. Slik vil utføringen av løkken fortsette inntil betingelsen ikke lenger er oppfylt.

Med `while`-setningen vil betingelsen testes før løkkekroppen eventuelt utføres. Det betyr at dersom betingelsen *ikke* er oppfylt ved inngangen til løkken, vil ikke løkkekroppen bli utført i det hele tatt.

Figur 2.15 *Utføring av en `while`-løkke*

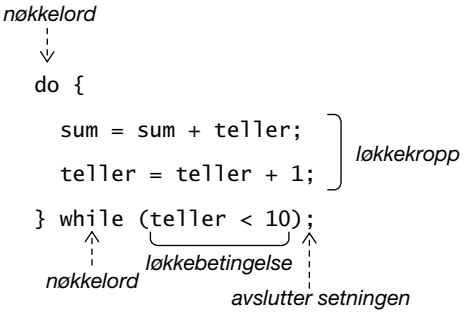


Løkke med test etter løkkekroppen

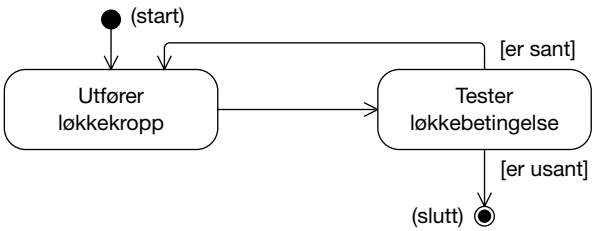
En `do-while` løkke i Java evaluerer løkkebetingelsen etter at løkkekroppen er utført. Figur 2.16 viser syntaksen for denne typen løkke. Løkkebetingelsen følger på slutten av løkken, og hele løkkekonstruksjonen *må* avsluttes med semikolon (`;`).

Hvis vi bruker `do-while`-setningen, testes betingelsen alltid *etter* at løkkekroppen er utført. Det vil si at programmet alltid vil utføre setningen(e) i en `do-while`-løkke *minst én gang*, slik Figur 2.17 viser.

Figur 2.16 *Eksempel på `do-while`-løkke*



Figur 2.17 *Utføring av en `do-while`-løkke*



Evig løkke

Under utføring av løkkekroppen må det være en handling som påvirker løkkebetingelsen slik at denne ikke blir oppfylt når den blir testet. Ellers risikerer vi nemlig en *evig løkke*, det vil si at utføringen av løkkekroppen aldri blir avsluttet fordi betingelsen forblir sann. På de fleste plattformer vil tastekombinasjonen Ctrl-C være nødvendig for å avbryte utføringen av programmet og dermed den evige løkken.

Eksempler på evige løkker er gitt i avsnittet "Nyttig å vite om løkker" på side 80.

Løkkebasert summering av heltall

Ved hjelp av løkker kan vi lage dialoger der bruker for eksempel kan oppgi et vilkårlig antall heltall som programmet skal summere.

Linje (1) i Program 2.17 leser inn et heltall som senere brukes til å kontrollere hvor mange ganger programmet skal repetere innlesing av et nytt tall som adderes til de foregående. Innlesingen av heltall som skal summeres gjøres i en `while`-løkke som starter i (2) og slutter i (3). Betingelsen `antallLest < antallTall` evalueres i (2) før løkkekroppen eventuelt utføres med innlesing av neste tall, oppdatering av summen og antallet tall som er lest så langt. Etter løkken skrives den endelige summen av de innleste tallene ut i (4).

Program 2.17 *Summering av heltall ved hjelp av løkker*

```
import java.util.Scanner;
class HeltallsSummering {
  public static void main(String[] args) {
    int antallTall,antallLest; // antall tall man hhv. skal og har lest inn
    int nesteTall,sum;         // neste tall og summen av alle innleste tall så langt

    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi antall heltall som skal summeres: ");
    antallTall = tastatur.nextInt();                                  // (1)
    tastatur.nextLine();

    antallLest = 0;
    sum = 0;

    while (antallLest < antallTall) {                                 // (2)
      System.out.printf("Gi %d. tall [heltall]: ", antallLest+1);
      nesteTall = tastatur.nextInt();
      tastatur.nextLine();
      sum = sum + nesteTall;
      antallLest = antallLest + 1;
    }                                                                 // (3)

    System.out.printf("Summen av %d heltall er %d%n", antallLest, sum); // (4)
  }
}
```

Kjøring av programmet for å summere fire heltall:

```
Gi antall heltall som skal summeres: 4
Gi 1. tall [heltall]: 12
Gi 2. tall [heltall]: 34
Gi 3. tall [heltall]: 4
Gi 4. tall [heltall]: -2
Summen av 4 heltall er 48
```

2.12 Påstander

Vi kan kontrollere at programmet oppfyller våre forutsetninger ved et bestemt sted i koden ved å definere en *påstand* (eng. *assertion*) i form av et sannhetsuttrykk. Dette uttrykket defineres i en `assert`-setning, som evalueres under utføring av programmet. Dersom sannhetsverdien er `false`, skrives det ut en feilmelding og programmet avbrytes. Hvis sannhetsverdien av uttrykket er `true`, fortsetter utføringen rett etter `assert`-setningen. For eksempel kan vi kontrollere at heltallsvariabelen `bredde` har en positiv verdi på følgende måte:

```
assert bredde > 0.0;
```

Denne setningen vil føre til at programmet stopper dersom variabelen `bredde` ikke har en positiv verdi i det `assert`-setningen utføres.

For å gi mer informasjon om grunnen til at programmet stopper, kan vi gi en forklarende tekststreng sammen med sannhetsuttrykket, for eksempel slik:

```
assert bredde > 0.0 : "Bredden av rektangelet må være > 0.0";
```

Dersom variabelen `bredde` ikke har en positiv verdi i det `assert`-setningen ovenfor utføres, vil programmet stoppe slik denne kjøringen viser:

```
> javac FlyttallsAreal3.java
> java -ea FlyttallsAreal3
Gi lengden av rektangelet [flyttall]: 12,5
Gi bredden av rektangelet [flyttall]: -2,5
Exception in thread "main" java.lang.AssertionError:
    Bredden av rektangelet må være > 0.0
        at FlyttallsAreal3.main(FlyttallsAreal3.java:16)
```

Utskriften gir informasjon om hvor påstanden er definert (`main()`-metoden i `FlyttallsAreal3`-klassen), kildekodefilen (`FlyttallsAreal3.java`) og linjenummer i denne (16). Merk bruken av flagget `"-ea"` ved kjøring. Dette slår *på* sjekking av påstander som er definert i programmet (se vedlegg F).

Program 2.18 bruker `assert`-setninger i (1) og (2) til å *kontrollere verdier før de brukes* i beregninger, og i (3) for å *kontrollere gyldigheten av resultatet*. Den andre av disse testene slår til dersom bredden av rektangelet har en negativ verdi, slik kjøringen ovenfor viser.

Program 2.18 *Bruk av påstander*

```
import java.util.Scanner;
class FlyttallsAreal3 {
  public static void main(String[] args) {
    double lengde;
    double bredde;

    Scanner tastatur = new Scanner(System.in);
    System.out.print("Gi lengden av rektangelet [flyttall]: ");
    lengde = tastatur.nextDouble();
    tastatur.nextLine();

    System.out.print("Gi bredden av rektangelet [flyttall]: ");
    bredde = tastatur.nextDouble();

    assert lengde > 0.0 : "Lengden av rektangelet må være > 0.0";             // (1)
    assert bredde > 0.0 : "Bredden av rektangelet må være > 0.0";             // (2)

    double grunnflate = lengde * bredde;

    assert grunnflate > 0.0 : "Rektangelets grunnflate kan ikke være negativ";// (3)

    System.out.printf(
      "Grunnflaten av et rektangel med lengde %.2f og bredde %.2f er %.2f%n",
```

```
        lengde, bredde, grunnflate);
    }
}
```

Desom vi legger beregningen av grunnflate inn i egen metode som starter før (1) og slutter etter (3), vil de to første testene kontrollere hhv. *forkrav* (eng. *precondition*), og den siste testen resultatet fra den, dvs. *etterkrav* (eng. *postcondition*). Dette er en anbefalt måte å bruke `assert`-setningen på. Vi bør teste at verdier som gis inn til en metode og at resultatet den returnerer er gyldige.

Vi kan gi et vilkårlig sannhetsuttrykk i en påstand, for eksempel:

```
assert bredde > 0 && lengde > 0;
```

Her kunne vi også ha gitt en forklarende tekststreng som skulle skrives ut dersom uttrykket ble evaluert til `false` under utføring av programmet:

```
assert bredde > 0 && lengde > 0 : "Sidene i rektangelet må være > 0";
```

Ovenfor har vi kun sjekket verdien til to variabler. Sannhetsuttrykket i en påstand kan også inneholde aritmetiske uttrykk og kall til metoder, men disse må ikke ha sideeffekter. Det vil si at de verken må endre verdien til lokale variabler eller tilstanden til objekter i programmet. Ellers vil utføringen av programmet bli avhengig av de påstandene som defineres.

2.13 KONTROLLSPØRSMÅL

Spørsmål 2.1 Hva blir skrevet ut i terminalvinduet fra metodekallene under?

(a) `System.out.println("10+10 er " + 20);`

(b) `System.out.println("10+10 er " + 10 + 10);`

(c) `System.out.println(10 + 10 + 20);`

Spørsmål 2.2 Forklar begrepet variabel. Hva er en lokal variabel?

Spørsmål 2.3 Skriv en deklarasjon for en heltallssvariabel med navn `antallPunkter`, som samtidig initialiserer den til verdien 35. Hva kalles en verdi som skrives direkte i kildekodefilen?

Spørsmål 2.4 Hva er en konstant? Modifiser deklarasjonen fra spørsmål 2.3 slik at variabelen blir en konstant.

Spørsmål 2.5 Hvilke av følgende variabler har gyldige navn? Hvilke navn er fornuftige? Begrunn svarene.

(a) `minste-Pris`

(b) `minstePris`

(c) `XYZ`

(d) `xKoordinat`

(e) `y2k`

(f) `erOppfylt`

(g) `antallDagerIFebruarIEtSkuddÅr`

(h) `JDK_1_5_0`

Spørsmål 2.6 Gi en definisjon av begrepet datatype. Hva er en primitiv datatype?

Spørsmål 2.7 Fyll ut det som mangler i setningene nedenfor:

(a) Et aritmetisk uttrykk kan bestå av operander og _____.

(b) Multiplikasjon (`*`) krever __ operander, og er derfor en _____ operator.

(c) Operatoren `-` i uttrykket `-4` er en _____ operator, mens – i uttrykket `5 - 4` er en _____ operator.

Spørsmål 2.8 Operandene i et uttrykk i Java evalueres alltid fra _____ mot _____. Dersom to regneoperatorer med forskjellig _____ står etter hverandre i et uttrykk, vil operatoren med _____ _____ bli utført først.

Spørsmål 2.9 Dersom to operander med samme presedens står rett etter hverandre i et uttrykk, brukes _____ til å avgjøre hvilken operator som skal utføres først. Operatoren – (unær minus) er _____ og grupperer fra _____ mot _____, mens operatoren – (subtraksjon) er _____ og grupperer fra _____ mot _____.

Spørsmål 2.10 Evaluer følgende uttrykk. Begrunn rekkefølgen operasjonene utføres i for hvert tilfelle.

(a) `3 + 2 - 1`

(b) `2 + 6 * 7`

(c) `- 5 + 7 - - 6`

(d) `2 + 4 / 5`

(e) `5 * 2 - - 3 * 4`

(f) `10 / 0`

(g) `2 + 4.0 / 5`

(h) `10 / 0.0`

(i) `4.0 / 0.0`

(j) `2 * 4 % 2`

Spørsmål 2.11 Bruk `System.out.printf()`-metoden til å skrive ut følgende:

(a) Et sekssifret heltall der fortegnet skal vises, f.eks. 123456 som +123456.

(b) Flyttallet 123456789,3837 med vitenskapelig notasjon, dvs. som 1.234567e+08.

(c) Strengen `"Vi satser 100% på å lære oss Java!"`.

(d) Tallet 1024 som et høyrejustert 8-sifret heltall, dvs. som 00001024.

Spørsmål 2.12 Valg- og repetisjonssetninger er to former for _____. Disse lar oss _____.

Spørsmål 2.13 Et _____uttrykk kan ha kun en av to verdier, _____ eller _____. Verdien av et slikt uttrykk kan vi tilordne til en _____ variabel.

Spørsmål 2.14 Hvilke av følgende tilordninger er tillatt i Java, gitt at variabelen `test1`, `test2` og `test3` alle har type `boolean`? Begrunn svarene.

(a) `test1 = 2 < 3;`

(b) `test1 = 2 + 3 < 1 - - 5;`

(c) `test1 = "true";`

(d) `test2 = false;`

(e) `test3 = test1;`

Spørsmål 2.15 Java definerer en rekke sammenligningsoperatorer, som blant annet kan brukes i _____uttrykk. For å teste om to verdier er like, kan vi bruke operatoren __. For å teste om to verdier er ulike, kan vi bruke operatoren ____.

Spørsmål 2.16 Hva blir verdien av uttrykket `b1 && !b2 || !b1 == b3`, gitt at de boolske variabelene `b1` , `b2`, `b3` har verdien `true`, `false`, `true`, henholdsvis?

Spørsmål 2.17 Hva blir verdien til variabelen `test` i linje (1)–(4)? Er alle tilordningene tillatt i Java?

```
boolean test;
int antallMenn = 12;
int antallKvinner = 7;
test = antallMenn < antallKvinner;       // (1)
test = antallMenn = antallKvinner;       // (2)
test = antallMenn * antallKvinner >= 90; // (3)
test = 2 * antallKvinner < antallMenn;   // (4)
```

Spørsmål 2.18 Vi kan kombinere sannhetsuttrykk ved hjelp av _____ operatorer. Java har tre slike operatorer: __ (negasjon), __ (betinget OG), og __ (betinget ELLER).

Spørsmål 2.19 Hva blir verdien av følgende sannhetsuttrykk, når verdien av de boolske variablene `b1`, `b2` og `b3` er h.h.v. `true`, `false` og `true`:

(a) `b3 || 3 * 0.5 < 2 + 3 * -1`

(b) `4 + 1 == 8 - 3 && b2`

(c) `5 < 4 || !b2`

(d) `b1 && !b2 || b3`

(e) (b2 || 2 < 3) && (!b3 || b1)

(f) ! ((b1 && b3) || (b2 != b1))

Spørsmål 2.20 Bruk De Morgans lover til å omforme følgende sannhetsuttrykk, når `b1` og `b2` er boolske variabler.

(a) `!(b1 || b2)`

(b) `!(b1 == b2 && b2)`

(c) `!(!b2 || b1 == true)`

(d) `(!b1 == b2 || b2)`

(e) `(!(b1 == b2) || !(b2 == b1))`

Spørsmål 2.21 Lag en valgsetning som skriver ut om heltallsvariabelen `antallPunkter` er et partall eller et oddetall. (Partall er tall som kan deles med 2 uten restledd.)

```
if (antallPunkter % 2 == 0)
  System.out.println("antallPunkter (" + antallPunkter + ") er et partall.");
else
  System.out.println("antallPunkter (" + antallPunkter + ") er et oddetall.");
```

Spørsmål 2.22 Et annet ord for repetisjonssetning er _____. Kriteriet for å gjenta _____ kalles _____.

Spørsmål 2.23 Forklar forskjellen på de to typene løkker som er brukt i dette kapitlet. Hvilken type ville du bruke for å skrive ut alle partall mellom en nedre og en øvre grense?

Spørsmål 2.24 En `assert`-setning spesifiserer et _____uttrykk, og kan i tillegg angi en forklarende _____.

Spørsmål 2.25 Hvilke utsagn er sanne?

(a) Påstander blir alltid utført under kjøring.

(b) Dersom sannhetsuttrykket i en påstand er sant, vil utføringen stoppe.

(c) Dersom sannhetsuttrykket i en påstand er usant, vil utføringen stoppe.

Spørsmål 2.26 Program 2.15 leser inn antall timer en person har arbeidet i inneværende uke og beregner ukelønnen til vedkommende, men uten å kontrollere at timetallet som leses inn er gyldig. Skriv to påstander som kontrollerer antallet timer før valgsetningen ved (1) utføres, slik at brukeren ikke kan gi inn et negativt tall eller et timetall som er større enn det som faktisk er mulig i en uke.

2.14 OPPGAVER

Oppgave 2.1 Lag et program som beregner temperaturen i grader fahrenheit fra en gitt temperatur i grader celsius. Formel for omregning mellom de to enhetene er: fahrenheit = (9.0 * celsius) / 5.0 + 32.0. La i første omgang programmet gjøre beregninger for en bestemt temperatur, f.eks. 20 grader celsius.

Oppgave 2.2 Utvid programmet fra oppgave 2.1 slik at det også regner ut tilsvarende temperatur i grader kelvin, ved hjelp av følgende formel: kelvin = celsius + 273.16.

Oppgave 2.3 Utvid så programmet slik at temperaturen leses inn fra tastaturet.

Oppgave 2.4 Et firma gir selgerne sine lønn etter et provisjonsbasert system. Selgere får 200 kr per uke pluss 9 prosent av summen av prisene til produktene selgeren har solgt. For eksempel vil en selger som har solgt produkter for 5000 kr, motta 200 kr pluss 9 prosent av 5000 kr, det vil si 650 kr.

Prisene på produktene er som følger:

(a) Produkt A: 239,99

(b) Produkt B: 129,75

(c) Produkt C: 99,95

(d) Produkt D: 350,89

Lag et program som leser inn antall produkter av hver type en selger har solgt i den foregående uken, og deretter beregner og skriver ut lønnen til selgeren. Programmet skal skrive ut en pen oppstilling der antall solgte enheter samt stykkpris vises for hver vare. Bruk påstander til å kontrollere at antallet produkter som leses inn er positivt.

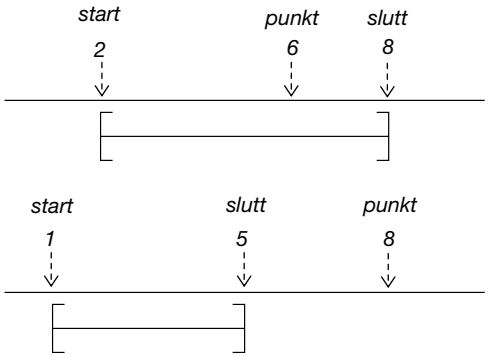
Oppgave 2.5 Skriv et program som leser lengden av en tidsperiode i sekunder, og så skriver dette ut i timer, minutter og sekunder. Hvis for eksempel brukeren skriver 3603, skal programmet skrive:

```
3603 sekunder = 1 time, 0 minutter og 3 sekunder
```

Sørg for at programmet gir ut grammatisk korrekte utskrifter, det vil si 0 timer, 1 time, 2 timer, o.s.v. Du kommer til å trenge valgsetningen for å få dette til. Bruk påstander til å kontrollere at tidsperioden som leses inn er positiv.

Oppgave 2.6 Lag et program som finner ut om en verdi er innenfor eller utenfor et intervall. For eksempel er verdien 6 innenfor intervallet som går fra 2 til 8, mens verdien 8 er utenfor intervallet som

går fra 1 til 5 (se figuren under).



Lag et program som

1. leser inn start- og sluttverdier for et intervall
2. leser inn verdien til et punkt
3. rapporterer tilbake om punktet lå innenfor eller utenfor intervallet

Her er et eksempel på hvordan kjøringen av programmet kan se ut:

```
Skriv inn startverdi: 2
Skriv inn sluttverdi: 8
Skriv inn verdi for punkt: 6
Punktet med verdi 6 ligger innenfor intervallet fra 2 til 8.
```

Hva synes du burde skje hvis brukeren gir inn en startverdi som er større en sluttverdien? Vurder å bruke påstander for å kontrollere at innleste verdier definerer et gyldig intervall.

Hva synes du programmet burde rapportere hvis punktverdien ligger akkurat på grensen til intervallet? Få programmet ditt til å oppføre seg på den måten som du synes er best.

Oppgave 2.7 Et program skal finne ut hvor et punkt gitt ved koordinater `x` og `y` ligger i forhold til origo. Dersom for eksempel `x` har verdien 2 og `y` har verdien 3, skal programmet skrive ut

```
Punktet (x=2,y=3) ligger i øvre, høyre kvadrant.
```

Har `x` verdien –3 og `y` verdien –1, skal utskriften være

```
Punktet (x=-3,y=-1) ligger i nedre, venstre kvadrant.
```

Skriv et program som leser inn en x- og en y-koordinat og som skriver ut en setning som sier hvilken kvadrant punktet (x,y) ligger i.

Oppgave 2.8 Skriv et program som leser inn og summerer *n* heltall vha. en `do-while`-løkke. Programmet skal ha samme funksjonalitet som Program 2.17.

Oppgave 2.9 Lag et program som leser inn en sekvens av positive heltall fra tastaturet og skriver ut en

rapport som forteller gjennomsnittet av disse tallene. Anta at sekvensen avsluttes med et negativt tall. Hvis brukeren for eksempel skriver

```
23
42
53
3
54
82
37
43
29
44
-1
```

skal programmet skrive ut en rapport som gir gjennomsnittet av disse tallene (ikke inkludert den avsluttende negative verdien), for eksempel

```
Summen av tallene er 410.
Gjennomsnittet av de 10 tallene er 41.
```

Oppgave 2.10 Anta at brukerne også ønsker å beregne gjennomsnittet av negative tall. Nå kan vi ikke lenger teste på tallverdien for å finne ut når det ikke er flere tall som skal tas med i beregningen. I stedet kan vi forutsette at antallet tall er kjent og la brukeren starte med å oppgi dette. Hvis brukeren for eksempel oppgir at det er 7 tall og så skriver tallene

```
-3,25
-4,66
2,14
3,66
3,78
3,67
-4,33
```

skal programmet skrive ut en lignende rapport som i oppgave 2.9, for eksempel

```
Summen av tallene er 1,01
Gjennomsnittet av de 7 tallene er 0,14
```

La programmet skrive ut en forklaring på programmet først, og spør så etter antallet tall før innlesingen av tallverdiene starter. Du kan anta at det alltid gis minst ett tall (men programmet bør kontrollere dette).

Oppgave 2.11 Lag et program som tilordner karakter A-F basert på den poengsummen en student har oppnådd på eksamen. Grenser for de ulike karakterene er:

A: 91-100 poeng
B: 81-90 poeng
C: 61-80 poeng
D: 31-60 poeng
E: 15-30 poeng

Dersom en student får mindre enn 15 poeng, stryker vedkommende (karakter F).

La programmet lese inn studentens nummer og poengsum, der tillatte verdier for nummer er partall fra og med 1 til og med 200, og tillatte verdier for poengsum er fra 0,0 til 100,0. Bruk påstander til å kontrollere de verdier som leses inn.

En mulig kjøring av programmet kan være:

```
Oppgi studentnummer [heltall] : 20
Oppgi poengsum [flyttall] : 87
Student 20 får karakter B
```

dersom innlest studentnummer og poengsum er gyldige verdier, og

```
Oppgi studentnummer [heltall] : 24
Oppgi poengsum [flyttall] : 102,3
Exception in thread "main" java.lang.AssertionError:
Ugyldig poengsum for eksamen 102,3
Tillatte verdier fra 0,0 til 100,0
        at Karakter.main(Karakter.java:24)
```

dersom poengsummen som gis ligger utenfor tillatt intervall.