# Kapittel 5

# Din første klasse

**Læringsmål for dette kapitlet**

Etter å ha vært gjennom dette kapitlet skal du

- skjønne at objekter i et program er modeller av objekter i den virkelige verden
- kunne bruke og lage en enkel klasse med objektvariabler, konstruktører og metoder
- skjønne hvordan begrepene melding, klient og tjener avspeiles i programkode
- vite forskjellen mellom referansetyper og primitive datatyper
- vite hva modifikatoren final betyr for en referansetype
- vite hva det betyr at en referanse er null, og hvilke konsekvenser det kan få (NullPointerException)

Det finnes flere systematiske måter å løse et programmeringsproblem på. Tidligere (på 70- og 80-tallet) angrep man gjerne et slikt problem ved å fokusere på algoritmeabstraksjon. Man begynte med å sette opp en algoritme på et veldig høyt abstraksjonsnivå, eksempel: les inn data – gjør beregninger – skriv ut resultater. Deretter tok man for seg hvert punkt i denne algoritmen og detaljerte videre, ofte på flere nivåer avhengig av problemets kompleksitet. Det var ikke uvanlig at man på lavt nivå oppdaget at man hadde løst et problem tidligere. Å isolere programbiter som løste spesielle problemer, ble, og er fremdeles, til stor nytte i programutviklingen. Bibliotek med rutiner for eksempelvis matematiske beregninger, grafikk og håndtering av store datamengder i arkiv (databaser) er vanlig. Selv om dataene selvfølgelig er viktige, er det i slike bibliotek ikke noen helt klar sammenheng mellom dataene og de algoritmene som gjør noe med dataene.

Mange mener at en av årsakene til at det er så vanskelig å lage store og komplekse systemer, ligger i denne angrepsmåten. Man bør se data og algoritmer i sammenheng. Programvarebibliotek bør ikke bestå av bare behandlingsrutiner for data, men av en type moduler der disse rutinene er uløselig knyttet til de dataene som de kan behandle. Denne typen moduler kalles klasser. Java-API-et inneholder en mengde slike klasser.

Klasser bruker vi også når vi skal lage våre egne programsystemer for spesielle anvendelser. Vi kan ha klasser som håndterer bankkontoer, bedrifter eller idrettslag, eller omtrent hva som helst annet.

Å angripe et programmeringsproblem på denne måten kalles objektorientering. Objekter i den virkelige verden avspeiles i programkoden.

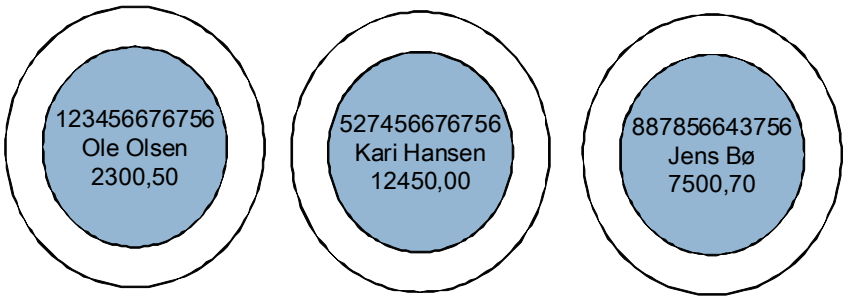
## 5.1 Objekter og klasser

Allerede etter å ha gjennomgått de første kapitlene i denne boka og løst noen av oppgavene øyner du kanskje muligheten for at et program raskt kan bli stort og omfattende. Vi skal nå se hvordan vi ved å dele opp i mindre deler kan få oversikt og kontroll over problemløsningen. Og da skal vi knytte sammen dataene og algoritmene på en mer direkte måte enn det vi har gjort hittil. Vi skal lage klasser.

Som eksempel skal vi se på et program som holder orden på saldoen på en bankkonto. Vi gjør det så enkelt at vi kun teller saldoen opp og ned, vi lagrer ikke alle enkelttransaksjonene. Du vil neppe ha problemer med å lage et lite program som leser inn transaksjoner og fortløpende oppdaterer en saldo. Men tenk deg at du skal utvide programmet med et par kontoer til. Og det er ønskelig at utskriftene inneholder kontoens navn og nummer. Det er fortsatt overkommelig, men kompleksiteten øker raskt nå. Etter at programmet har vært i drift en stund, får du lyst til å legge inn fødselsdatoen til kundene også. Du må endre koden flere steder.

Å lage kode som er enkel å endre og vedlikeholde, er viktig. For å få til det må koden være bygd opp av mindre enheter på en logisk måte. I dette eksemplet betyr det at vi lager en enhet (klasse) som beskriver det som gjelder alle kontoer. Hvis vi

nå vil legge til noe, vet vi hvor vi skal gjøre det. Vi utvider den felles beskrivelsen som gjelder *alle* kontoer.




Figur 5.1 Tre kontoobjekter med samme oppbygning, men forskjellig datainnhold

I den virkelige verden har vi kontoer, eller konto*objekter*, se figur 5.1. De er alle bygd på samme måten, her noe forenklet med kontonummer, navn og saldo. Dette er egenskapene, eller *attributtene*, til objektene.

objekt

attributt

Hva gjør vi med kontoer i den virkelige verden? Vi sjekker saldoen, og vi setter inn og tar ut penger. Noe avhengig av omstendighetene varierer det hvordan vi gjør dette. Vi kan gå i bankfilialen og få godskrevet en sjekk på kontoen vår, eller vi kan ta ut et beløp fra en minibank. Uansett hvordan vi gjør dette, så er vi den aktive parten og kontoen den passive parten. Ingen vil finne på å si at kontoen selv utfører transaksjonen.

Imidlertid er dette en talemåte vi nå må venne oss til: Et kontoobjekt i et program utfører selv transaksjonene, dog etter at det har fått beskjed om å gjøre det. Alle oppgaver som vi normalt vil gjøre i forhold til en konto, lar vi kontoobjektet selv få ansvaret for i et program. Slik er det i alle sammenhenger i objektorientert programmering. Objektene, enten de i den virkelige verden er levende eller døde, lever et liv i programmet vårt, de har mye kunnskap, og de har ansvaret for å utføre mange forskjellige oppgaver.

Oppgavene løses ved algoritmer som er uavhengig av datainnholdet i hvert enkelt objekt. Algoritmene formuleres generelt i forhold til attributtene.

Vi bruker gjerne begrepet *operasjon* om oppgaver som objekter har ansvaret for å utføre. En *klasse* er en formell beskrivelse av en mengde objekter med samme attributter og samme operasjoner. (I kapittel 2 definerte vi en klasse fra et kodeståsted som en logisk samling deklarasjoner. Vi skal snart se sammenhengen mellom de to definisjonene.)

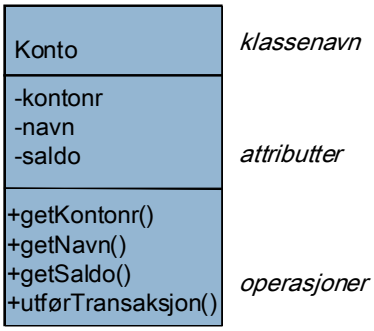
operasjon

klasse

## Steg 1: Fra tekstlig beskrivelse til klassediagram

Vi kan sette opp følgende oversikt for problemet vi skal løse:

Klasse: konto.
Attributter: kontonummer, navn, saldo.
Operasjoner: Finn saldo, navn og kontonummer. Utfør transaksjoner.




Figur 5.2 Første skritt på veien fra virkeligheten til programdesign (UML)

klasse-diagram

*Klassediagram* brukes på veien fra en tekstlig beskrivelse av en klasse og fram til programkoden. Figur 5.2 viser første skritt på denne veien. På samme måte som aktivitetsdiagrammene fra kapittel 3 og 4 så er klassediagram en del av UML (Unified Modeling Language). Operasjoner skilles fra attributter ved en strek. I tillegg er det vanlig å sette () etter navnene på operasjonene.

- og + i klasse-diagram

På figuren har attributtene minus (-) foran seg, mens operasjonene har pluss (+) foran. Minus betyr "gjemt inne i objektet", mens pluss betyr tilgjengelig utenfra. Et objekt i et program har altså dataene gjemt inne i seg, mens operasjonene er tilgjengelig utenfra, slik at andre kan be om å få dem utført.

get-operasjoner

La oss se litt nærmere på operasjonene. Vi ser at vi i tillegg til operasjonen utførTransaksjon() har noe som heter getKontonr(), getNavn() og getSaldo(). Dette er operasjoner for å hente ut attributtverdiene som er gjemt inne i objektet. Merk at vi velger å bruke den engelske forstavelsen get- selv om vi for øvrig skriver norsk – dette fordi denne forstavelsen for å hente ut en attributtverdi i praksis er standard i Java-miljøet. Det er for eksempel slik at mange av verktøyene en bruker i forbindelse med programutvikling, forutsetter at en bruker forstavelsen get- i slike sammenhenger.

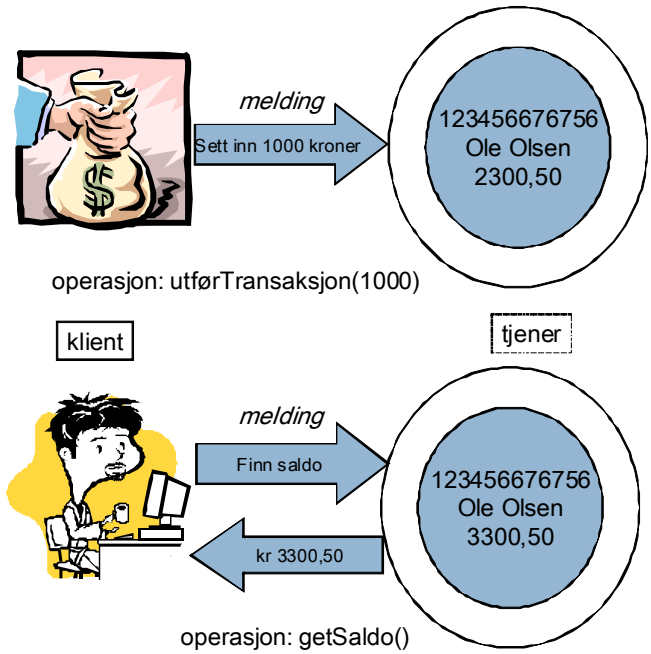
## Steg 2: Å bruke klassen Konto

Det er nå du egentlig trenger å lære mange ting på én gang. Og det er jo ikke mulig, vi skal ta det hele stegvis, men det kan nok hende du etter hvert trenger å gå tilbake noen sider og lese ting om igjen.

Du trenger å lære å kode klassen, det omfatter flere nye og vanskelige begrep og teknikker. Men en klasse i seg selv har ingen verdi, du må samtidig lære å bruke den. Du må altså lære å lage og bruke objekter i programkoden. Det er først da du

får et program som kan gjøre noe nyttig. En analogi er matoppskrifter. Du kan lese oppskriften så mange ganger du vil, men det blir ingen mat i magen uten at du bruker oppskriften til å lage noe.

Vi velger å begynne med å bruke klassen. Det er ofte den måten vi arbeider på. Vi finner ut hva objektene våre trenger å kunne gjøre, og så lager vi klassen på basis av det.




Figur 5.3 Bruk av objekt med data om Ole Olsens konto

Se figur 5.3, der vi jobber med Ole Olsens konto. Objektet er tegnet som en ring. Det symboliserer at dataene er gjemt inne i objektet. Ringen er objektets grense mot omverdenen. Den består av operasjonene med + til høyre på figur 5.2 (selv om det ikke er vist på figuren). Vi kaller dette objektets *grensesnitt* mot omverdenen.

grensesnitt

Vi sier at omverdenen sender *meldinger* til objektet. Objektet, som utfører operasjonen, spiller rollen *tjener* i forhold til den som sender meldingen. Sistnevnte kalles *klient*.

melding
tjener
klient

På figuren sender klienten to meldinger til tjeneren. Den første ber om at 1000 kroner settes inn på kontoen, og dataverdien 1000 sendes derfor sammen med meldingen. Den andre meldingen etterspør saldoen. Da sender objektet et svar tilbake.

Vi ser altså at vi i noen tilfeller må sende data sammen med meldingene. Andre ganger får vi data tilbake. Vi kan også tenke oss kombinerte meldinger. Eksempelvis kan det være aktuelt at en operasjon som skal ta penger ut av en konto, rapporterer tilbake om det var dekning på kontoen.

**Programliste 5.1 EnKonto**

```
1   /**
2    * EnKonto.java
3    *
4    * Eksempel på bruk av klassen Konto.
5    * For at dette programmet skal kompilere, må du også ha filen Konto.java
6    * tilgjengelig i samme mappe, eventuelt ta den inn i "prosjektet" ditt.
7    */
8   class EnKonto {
9     public static void main(String[] args) {
10
11      /* Vi oppretter et objekt av klassen Konto. Objektet heter olesKonto */
12      Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
13      olesKonto.utførTransaksjon(1000.0); // setter inn 1000 kroner
14      double saldo = olesKonto.getSaldo(); // spør objektet om den nye saldoen
15      System.out.println("Etter innskudd er saldoen lik " + saldo);
16    }
17  }
18
19  /* Utskrift:
20  Etter innskudd er saldoen lik 3300.5
21  */
```

klient-program

Programliste 5.1 viser et program som oppretter et Konto-objekt og utfører operasjonene på figur 5.3. Denne main()-metoden fungerer som klient og kalles derfor ofte *klientprogram*. (Merk at du må ha klassen Konto for å kompilere og kjøre dette eksemplet. Du finner den på filen *Konto.java*. Se hodekommentaren.)

Vi skal gjennomgå programmet linje for linje. Før vi kan gjøre noe med et objekt, må vi lage det. Og det skjer i linje 12:

```
Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
```

referanse-type
referanse
new
konstruktør

På venstre side av likhetstegnet gjør vi egentlig noe som vi har gjort før: Vi deklarerer en variabel som får navnet olesKonto. Datatypen er en klasse. Slike datatyper kaller vi *referansetyper*. Variabler av referansetyper kaller vi *referanser*. Det som står på høyre side av likhetstegnet, er imidlertid langt mer komplisert enn det vi er vant til. For å lage et objekt må vi bruke det reserverte ordet new sammen med noe som kalles en *konstruktør*. En klasse tilbyr en eller flere konstruktører, slik at det er mulig å lage objekter av klassen.

Klassen Konto tilbyr en konstruktør der vi må sette opp kontonummer (datatypen long), navn og startsaldo i rekkefølge. Etter at linje 12 er utført, har vi altså et objekt som vi kan sende meldinger til, det vil si at vi kan be objektet om å utføre operasjoner.

metode

Operasjoner på objekter blir til *metoder* når vi bruker dem i et program.

I linje 13 i programmet skal vi sette inn 1000 kroner:

```
olesKonto.utførTransaksjon(1000.0);
```

Vi setter opp transaksjonsbeløpet i parentes etter metodenavnet.

I linje 14 skal vi finne saldoen. Vi bruker nok en metode:

```
double saldo = olesKonto.getSaldo();
```

Her skal vi ha data ut av metoden, og den er laget slik at vi kan ta imot denne dataverdien i en variabel som vi kaller saldo. Merk at vi må sette opp parentes etter metodenavnet selv om vi ikke sender inn data.

## Oppgave

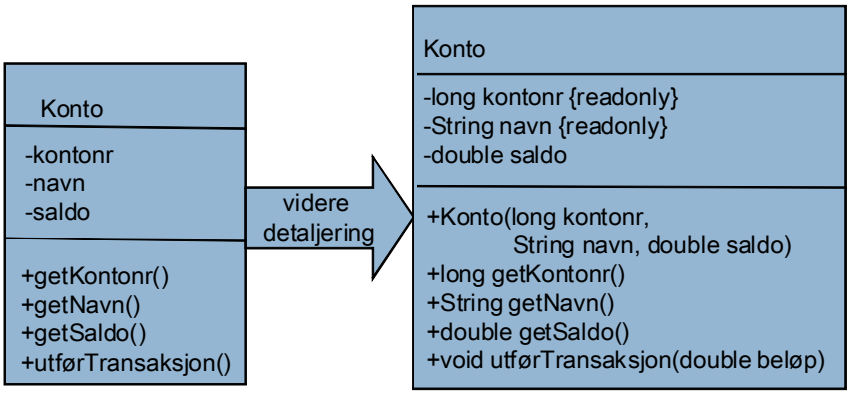
Transaksjonsbeløpet kan være lagret i en variabel. Variabelen må være av heltalls- eller desimaltallstype. Hvis variabelen heter transaksjon, skriver vi

```
olesKonto.utførTransaksjon(transaksjon);
```

Endre programliste 5.1 slik at programmet går i løkke og leser inn transaksjoner fra brukeren. Den nye saldoen skal for hver gang skrives ut i kommandovinduet. Hvis saldoen blir negativ, skal en advarsel skrives ut i en meldingsboks.

### Steg 3: Å detaljere klassediagrammet

Nå er vi kommet så langt at vi skal lage klassen Konto. Vi jobber skrittvis og begynner med å detaljere klassediagrammet.




Figur 5.4 Detaljering av klassen Konto for programmering i Java (UML)

Studer figur 5.4. Alle attributtene har fått datatyper. Noen av attributtene skal ikke kunne endres etter at objektet er laget, de har fått merkelappen {readonly}.

Det har også skjedd noe med operasjonene. Eksempelvis er getSaldo() blitt til double getSaldo(). Det er for at den *skal passe sammen med bruken,* slik vi har sett det i programliste 5.1:

```
double saldo = olesKonto.getSaldo();
```

returtype
returverdi

Når en klient sender meldingen getSaldo() til et objekt, får den en verdi av typen double tilbake. Vi sier at double er *returtypen* til metoden getSaldo(). Verdien som returneres, kalles *returverdien*. En metode kan aldri returnere mer enn én verdi, men denne verdien kan være et objekt.

Tilbake til figur 5.4. Operasjonen utførTransaksjon() er blitt til void utførTransaksjon(double beløp). Denne metoden har vi brukt på følgende måte:

```
olesKonto.utførTransaksjon(1000.0);
```

argument
parameter

Tallet 1000.0 passer med double beløp. Data som sendes inn til et objekt, kalles *(aktuelle) argumenter*. Her er tallet 1000.0 et slikt argument. Variablene som tar imot argumentene, kalles *parametere*.[11] Her er beløp en slik parameter. En metode kan ha flere parametere. Da må det være samsvar i rekkefølge og type mellom argumenter og parametere.

Klientprogrammet får ikke data ut av denne metoden, derfor skriver vi void foran metodenavnet.

Konstruktøren er også kommet med i klassediagrammet nå:

```
Konto(long kontonr, String navn, double saldo)
```

Merk at rekkefølgen av dataene i parentesen også her er i samsvar med bruken:

```
Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
```

Vi bruker begrepene parameter og argument også i forbindelse med konstruktører.

Tabell 5.1 viser en oversikt over argumenter, parametere, returtyper og returverdier i kommunikasjonen mellom klientprogrammet i programliste 5.1 og objektet olesKonto.

---

11. Vi bruker ordene parameter og argument slik de brukes i Javas språkspesifikasjon [Gosling, Joy, Steele 2005] og i UML [Rumbaugh, Jacobson, Booch 2005]. I annen litteratur vil man kunne finne uttrykkene aktuell og formell parameter eller aktuelt og formelt argument. Det "aktuelle" er det vi kaller argument, det "formelle" er det vi kaller parameter.

**Tabell 5.1 Sammenheng parametere–argumenter i programliste 5.1**

| Linje-nr. | Opera-sjon | Klientprogram (dataene eller *argumentene* som sendes til objektet) | Objektet (*parame-terne* som tar imot argu-mentene fra klienten) | Objek-tet, *retur-typen* | Objek-tet, *retur-verdien* |
|---|---|---|---|---|---|
| 12 | konstruk-tør | 123456676756L | long kontonr | ingen | ingen |
| | | "Ole Olsen" | String navn | | |
| | | 2300.50 | double saldo | | |
| 13 | utfør-Transak-sjon() | 1000.0 | double beløp | void | ingen |
| 14 | getSaldo() | ingen | ingen | double | 3300.5 |

## Steg 4: Å lage klassen Konto

Vi er nå klare til å se i detalj på koden i klassen Konto. Dette er den første klassen du ser, og det er selvfølgelig mye nytt.

Figur 5.5 viser sammenhengen mellom klassediagrammet og Java-koden. Merk nye begrep når vi forflytter oss fra klassediagrammet til Java-koden:

- Attributter blir variabler i klassen. Disse variablene kalles gjerne *objektvariabler*[12] på grunn av at hvert objekt av klassen har sitt eget sett med verdier. Merkelappen {readonly} blir final i Java-koden. Variabelen kan ikke forandre verdi etter at den har fått sin startverdi. Jamfør bruken av final her med bruken av final for for navngitte konstanter i kapittel 2 (side 57). objekt-variabel
- Operasjonene blir *objektmetoder* (eller bare "metoder") i klassen. objekt-metode
- *Medlemmene* i en klasse består av metodene og variablene som er deklarert i klassen. Konstruktørene er ikke å betrakte som medlemmer. Skopet til medlemmene i en klasse er hele klassekroppen. Utenfor skopet må vi *kvalifisere* navnet ved å angi hvor det er definert, eksempel: olesKonto.utførTransaksjon(500). Her er olesKonto en *kvalifikator*. medlem, kvalifisere, kvalifikator

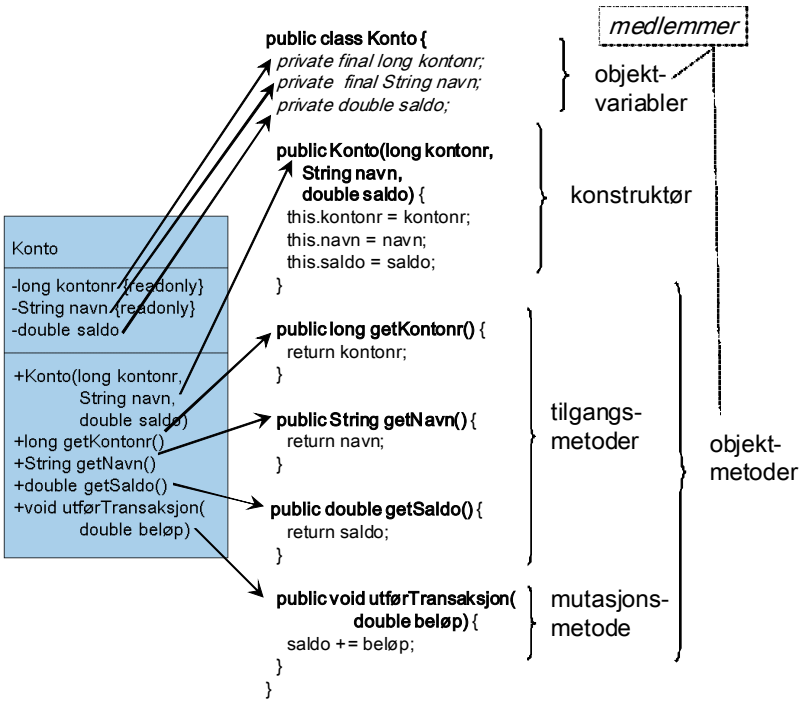
12. Instansvariabler er også et mye brukt ord, på samme måte som ordet instans ofte brukes om objekt. I stedet for å gi tilnærmet like definisjoner av disse ordene og i praksis bruke dem om hverandre, velger vi å bruke bare ordet objekt. Vi støtter oss på [Booch, Rumbaugh, Jacobson 2005, s. 175]: "The terms *instance* and *object* are largely synonymous, so for the most part, they may be used interchangeably."

tilgangs-metode

mutasjons-metode

mutabel/ immutabel klasse

- Vi skiller mellom tilgangsmetoder og mutasjonsmetoder. *Tilgangsmetoder* (engelsk: accessors) henter ut data fra et objekt eller gjør beregninger basert på disse. De forandrer ikke på datainnholdet. Metoder som endrer datainnholdet, kalles *mutasjonsmetoder* (engelsk: mutators). Mange klasser tilbyr bare tilgangsmetoder, det vil si at det ikke er mulig for klienten å endre på datainnholdet. Slike klasser kalles *immutable klasser*. Klasser som tilbyr mutasjonsmetoder, kalles *mutable klasser*.




Figur 5.5 Fra klassediagram til Java-kode

private
public

I klassediagrammet har vi minus- og plusstegn. Minustegnet fra klassediagrammet blir private i koden. Det betyr at variablene blir gjemt inne i objektene. (Her er private brukt for variablene, men det kan også brukes for metoder.) Plusstegnet i klassediagrammet blir public i koden, det vil si tilgjengelig for verden utenfor objektet. Konstruktøren må være offentlig for at det skal være mulig å lage objekter av klassen. Metodene utgjør bindeleddet mellom dataene som er gjemt inne i objektet, og omverdenen, se figur 5.6.

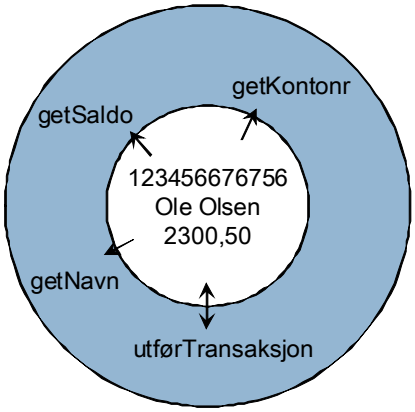
hode
innhold
kropp
implementere

Fete typer på figur 5.5 viser klassens *hode* samt konstruktørens og metodenes *hoder*. Når vi programmerer *innholdet* (eller *kroppen*) i en konstruktør eller en metode (magre typer på figuren), sier vi at vi *implementerer* konstruktøren eller metoden. Da kan vi bruke (de private) objektvariablene som er vist i *kursiv* på figu-

ren. *Klassekroppen* er hele klassen unntatt første og siste linje, det vil si hele den blokken som utgjør klassen.

Endelig nevner vi at et objekt er et eksempel på en *datastruktur*. En datastruktur er en generell betegnelse på en samling data som er lagret under ett navn i primærminnet. Som datastruktur er et objekt karakterisert ved at det kan inneholde ulike typer data, og vi får tilgang til de ulike dataene via navn. "Tabell" er en annen viktige form for datastruktur. Der er alle elementene av samme type, og de aksesseres via nummer. Tabeller introduseres i kapittel 7.

datastruktur




Figur 5.6 Metodene er bindeleddet mellom dataene inne i objektet og omverdenen

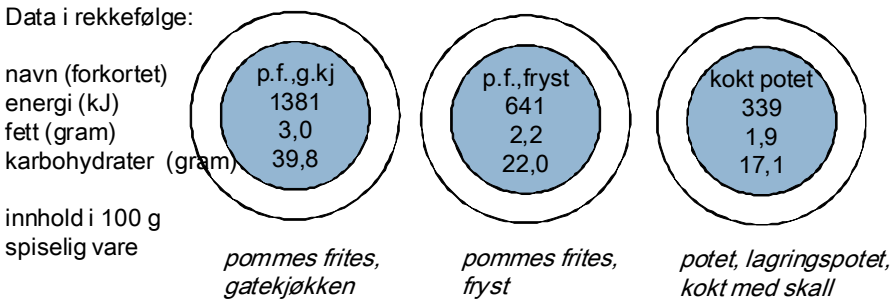
**Programliste 5.2 Konto**

```
1   /**
2    * Konto.java
3    *
4    * Klassen beskriver en bankkonto med nr., navn og saldo.
5    * Det er mulig å utføre transaksjoner mot kontoen.
6    */
7
8   class Konto {
9     private final long kontonr;
10    private final String navn;
11    private double saldo;
12
13    public Konto(long kontonr, String navn, double saldo) {
14      this.kontonr = kontonr;
15      this.navn = navn;
16      this.saldo = saldo;
17    }
18
```

```
19    public long getKontonr() {
20      return kontonr;
21    }
22
23    public String getNavn() {
24      return navn;
25    }
26
27    public double getSaldo() {
28      return saldo;
29    }
30
31    public void utførTransaksjon(double beløp) {
32      saldo += beløp;
33    }
34  }
```

## Oppgave

Figur 5.7 viser tre matvareobjekter med næringsinnhold i 100 gram spiselig vare. På *http://matportalen.no/* finner du lange lister med næringsinnhold for forskjellige matvarer. For ikke å få altfor omfattende kode begrenser vi oss til energi (kilojoule, kJ), fett og karbohydrater.




Dataene er hentet fra http://matportalen.no/

Figur 5.7 Tre matvareobjekter med næringsinnhold

Du skal lage klassen Matvare som kan brukes til å lage objektene vist på figuren. I Java-programmet skal objektene ha navnene pomFritGkj, pomFritFryst og koktPotet.

En klient skal kunne få utført følgende oppgaver ved å henvende seg til et av objektene:

- Finn matvarens navn.
- Finn antall gram fett i en bestemt mengde (gram) av matvaren.
- Finn antall gram karbohydrater i en bestemt mengde (gram) av matvaren.
- Finn energiinnholdet i kJ for en bestemt mengde (gram) av matvaren.
- Finn energiinnholdet i kcal (kilokalorier) for en bestemt mengde (gram) av matvaren. 1 kJ = 4,18 kcal.

Mengden matvare skal være data som klienten sender til objektet sammen med meldingen. Eksempel fra et klientprogram:

```
double fett = pomFritFryst.finnFett(350);
```

Etter at denne setningen er utført, skal variabelen fett ha innholdet 7,7 (= 2,2 / 100,0) * 350,0).

Gjør følgende:

1 Tegn klassediagram i to detaljeringsnivåer som på figur 5.4. I denne oppgaven er alle attributtene {readonly}.

2 Programmer klassen.

3 Lag et klientprogram som oppretter de tre objektene og lar brukeren få regnet ut næringsinnholdet (energi i kJ og kcal, fett og karbohydrater) for ulike mengder av de tre matvarene. Programmet skal være bygd opp slik:

```
Opprett de tre matvareobjektene
Les matmengde som beregningene skal utføres for (avslutt med Esc)
while ESc ikke trykket
    regn ut næringsinnhold for denne mengde av alle tre matvarene
    skriv ut resultatene
    les matmengde som beregningene skal utføres for (avslutt med Esc)
```

Hvis problemer oppstår, kan det hende du vil ha nytte av å lese neste delkapittel, som i detalj tar for seg sammenhengen mellom en klasse og bruken av den.

## 5.2 Mer om sammenhengen mellom klassen og bruken av klassen

Vi skal se på et mer omfattende klientprogram og følge programflyten i detalj mellom klientprogram og koden i klassen Konto. Programmet oppretter to kontoobjekter og overfører penger fra den ene til den andre kontoen.

**Programliste 5.3 ToKontoer**

```
1  /**
2   * ToKontoer.java
3   *
4   * Oppretter kontoobjekter for Per og Ole og overfører kr 1000 fra Ole til Per.
5   */
```

```
6
7   class ToKontoer {
8     public static void main(String[] args) {
9
10      /* Oppretter to objekter, ett for Ole og ett for Per */
11      Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
12      Konto persKonto = new Konto(223456676756L, "Per Hansen", 5000);
13
14      /* Overfører kr 1000 fra Ole til Per */
15      olesKonto.utførTransaksjon(-1000.0);
16      persKonto.utførTransaksjon(1000.0);
17
18      /* Henter ut de nye saldoene */
19      double saldoOle = olesKonto.getSaldo();
20      double saldoPer = persKonto.getSaldo();
21      System.out.println("Etter transaksjonen er Oles saldo lik " + saldoOle);
22      System.out.println("Etter transaksjonen er Pers saldo lik " + saldoPer);
23    }
24  }
25
26  /* Utskrift:
27  Etter transaksjonen er Oles saldo lik 1300.5
28  Etter transaksjonen er Pers saldo lik 6000.0
29  */
```

I gjennomgangen nedenfor refererer vi til to programlister:

- Metoden main() i programliste 5.3.
- Klassen Konto i programliste 5.2.

Programflyten styres fra main().

- Linje 11–12 i main(): Vi lager to kontoobjekter. Vi bruker konstruktøren og det reserverte ordet new. Vi sender argumenter til konstruktøren. Disse kobles til parameterne (linje 13 i klassen Konto) i konstruktøren. Følgende skjer bak kulissene:

  For objektet olesKonto:

  ```
  long kontonr = 123456676756L
  String navn = "Ole Olsen"
  double saldo = 2300.50
  ```

  For objektet persKonto:

  ```
  long kontonr = 223456676756L
  String navn = "Per Hansen"
  double saldo = 5000
  ```

skjule variabler

Parameterne har fått verdi, og de fungerer fra nå av som om de er lokale variabler inne i konstruktøren. Linje 14–16 i klassen Konto setter *objektets egne variabler* lik parameterne. Nå har det seg slik at objektets egne variabler har samme navn som parameterne. Parameterne *skjuler* (engelsk: hide) dermed objektvariablene, og eksempelvis har ikke setningen navn = navn; noen effekt.

Begge sider av tilordningstegnet refererer til parameteren. Vi kan imidlertid skille objektvariabel fra parameter ved å sette this foran objektvariablene:

```
this.kontonr = kontonr;
this.navn = navn;
this.saldo = saldo;
```

this er et reservert ord som refererer til det objektet vi til enhver tid arbeider med. Vi må også nevne at det ikke er nødvendig å bruke samme navn på parameterne som på objektvariablene i en konstruktør. Vi kan lage konstruktøren slik:

this

```
public Konto(long startKontonr, String startNavn, double startSaldo) {
  kontonr = startKontonr;
  navn = startNavn;
  saldo = startSaldo;
}
```

- Tilbake til klientprogrammet vårt. Etter at objektene er laget, er vi tilbake i main(), der neste trinn er å overføre penger fra den ene kontoen til den andre (linje 15–16). Vi sender meldingen utførTransaksjon(), først til objektet olesKonto, der vi tar ut kr 1000, deretter til objektet persKonto, der vi setter inn kr 1000. La oss se i detalj på hva som skjer idet den første meldingen blir utført:

  ```
  olesKonto.utførTransaksjon(-1000.0);
  ```

  Vi tar med oss argumentet -1000 til linje 31 i klassen Konto. Der settes parameteren beløp lik denne verdien, og beløp fungerer fra nå av som en lokal variabel inne i metoden. Deretter utføres denne setningen:

  ```
  saldo += beløp;
  ```

  Det vil si at variabelen saldo i objektet olesKonto får endret sin verdi med -1000. (Her kunne vi skrevet this.saldo, men det er vanlig å utelate this når det ikke er navnekonflikt.)
- Så er vi tilbake i main() igjen for å skrive ut saldoen på de to kontoene. Ettersom saldoen er gjemt i kontoobjektene, bruker vi metoden getSaldo() for å hente den ut. Vi forflytter oss fra linje 19 i main() til linje 27 i klassen Konto. Metoden getSaldo() inneholder setningen:

  ```
  return saldo;
  ```

  Det betyr at verdien til variabelen saldo sendes ut av objektet. I main() tar vi imot denne verdien og lagrer den i variabelen saldoOle:

  ```
  double saldoOle = olesKonto.getSaldo();
  ```

## Hva hvis det ikke er penger nok på kontoen?

Da ønsker vi at objektet skal fortelle oss det, se figur 5.8. Det kan vi gjøre ved at metoden utførTransaksjon() returnerer false dersom det ikke er dekning på kon-

toen, og true dersom uttaket gikk bra. (Husk at beløp er et negativt tall dersom vi tar penger ut av kontoen.)

```
public boolean utførTransaksjon(double beløp) {
  if (saldo + beløp >= 0) {
    saldo += beløp;
    return true;
  } else {
    return false;
  }
}
```

return-setningen

Hvis det er nok penger på kontoen (saldo + beløp >= 0), reduseres saldoen, og vi forteller klienten at det gikk bra. I motsatt fall forteller vi at det ikke gikk bra. I siste tilfelle er saldoen urørt. Merk at return-setningen fører til umiddelbart uthopp fra metoden. Setningene etterpå blir *ikke* utført.

Klientprogrammet kan ta i bruk svaret på denne måten:

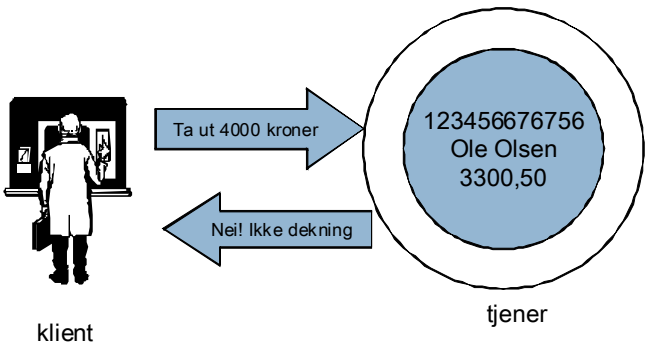
```
if (olesKonto.utførTransaksjon(-4000)) {
  System.out.println("Transaksjon ok");
} else {
  System.out.println("Transaksjon ikke ok");
}
```

Saldoen er kr 3300,50. Dermed blir utskriften:

```
Transaksjon ikke ok
```

Vi kan også skrive slik:

```
boolean okUttak = olesKonto.utførTransaksjon(-4000);
if (okUttak) {
  System.out.println("Uttak ok");
} else {
  System.out.println("Uttak ikke ok");
}
```




Figur 5.8 Hva om det ikke er dekning på kontoen?

## Oppgaver

1 Klassen Konto er som gitt i programliste 5.2. Finn eventuelle feil i følgende setninger:

```
arnesKonto = new Konto(123456789677, "Arne Jensen", 1000, 2000);
Konto johansKonto = new johansKonto(123456789677L, "Johan Hansen", 1000);
arnesKonto.getKontonr();
double saldo = arnesKonto.getSaldo(7000);
double beløp = arnesKonto.utførTransaksjon(1000);
Konto.utførTransaksjon(800);
```

2 Følgende utsagn eller Java-setninger er alle sammen feil. Forklar hva som er feil under hvert enkelt punkt:

a) Per Ås er en klasse.
b) void getLønn("Januar")
c) 1756 er en parameter.
d) Attributtet navn har et objekt person.
e) getLønn(String måned)
f) Argumentverdien kalles en parameter.
g) Attributter har hode og kropp.
h) En lokal variabel kan nås i alle metoder i klassen.

3 Rett opp feilene i følgende klasse

```
class Sirkel {
  final private radius;
  public sirkel(double radius) {
    radius = this.adius;
   public int beregnAreal() {
     return Math.PI * radius * radius;    // Math.PI er det samme som 3,14159 ...
   }
   public double beregnOmkrets() {
     omkrets = 2.0 * Math.PI * radius;
     return omkrets;
   }
}
```

4 Følgende klientprogram bruker klassen fra forrige oppgave. Fyll ut det som mangler (merket med --------), og rett opp eventuelle feil i resten av koden.

```
class Sirkelberegning {
  public static void main(String[] args) {
       -------- enSirkel = new --------;
       double arealet = enSirkel.--------;
       System.out.println("Arealet er lik " + arealet);
       -------- -------- = enSirkel.beregnOmkrets();
       System.out.println("Omkretsen er lik " + omkretsen);
   }
}
```

5 Finn ut hva som er lokale variabler, objektvariabler og parametere i programliste 5.2, side 155.

6 I oppgave 2 side 157 laget du klassen Matvare med klientprogram. Endre klassen slik at den lagrer næringsinnholdet per gram av varen, og ikke per 100 gram. Dette skal gjøres på en slik måte at du ikke behøver å forandre klientprogrammet.

## 5.3 Referanser og objekter

Vi skal se litt nærmere på sammenhengen mellom referanser og objekter. Vi tar utgangspunkt i følgende deklarasjon:

```
Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
```

referanser og objekter

Objektet er illustrert på figur 5.9. Variabelnavnet olesKonto står under noe som ligner på en pil. Det samme gjør navn, som er en referanse til et objekt av klassen String. Pilene skal vise at olesKonto og navn er referanser til objektene, og ikke objektene selv. Innholdet i en referansevariabel vil kunne ses på som adressen til det stedet i minnet der objektet er lagret. På denne måten kan vi si at en slik variabel "peker til" objektet.

String-objektet "Ole Olsen" er tegnet utenfor objektet olesKonto på figuren for å illustrere at objekter er selvstendige på den måten at det er mulig med flere referanser til det samme objektet.

Variabler av referansetyper brukes for å håndtere objekter. Slike variabler skiller seg fra variabler av primitive datatyper (int, long, double osv.) ved at de ikke inneholder dataene selv, men kun en referanse (peker) til dataene.

å lage String-objekter

Referansen navn peker altså til et objekt av klassen String. Denne klassen i Java-API-et er så mye brukt at det finnes en kortform for å lage objekter. Vi kan skrive for eksempel

```
String navn = "Ole Olsen";
```

i stedet for

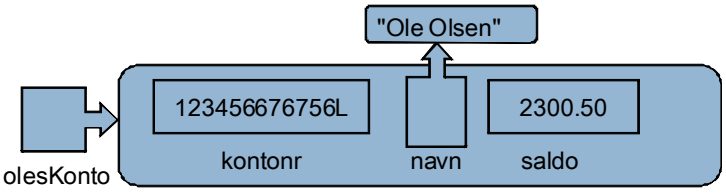
```
String navn = new String("Ole Olsen");
```

For øvrig opprettes alle objekter med det reserverte ordet new.

Vanligvis bruker vi for enkelhets skyld referansenavnene som om de var objektene de refererer til. Eksempelvis sier vi "objektet olesKonto" og ikke "objektet som olesKonto refererer til".

Det er ikke mulig å få tak i et objekt uten å gå via en referanse til objektet.

```
Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
```




Figur 5.9 Objektet olesKonto etter initiering

La oss lage to objekter:

```
Konto olesKonto = new Konto(123456676756L, "Ole Olsen", 2300.50);
Konto olesKonto2 = new Konto(223456676756L, "Ole Olsen", 0.0);
```

Objektene er vist øverst på figur 5.10. La oss sette den ene referansen lik den andre:

```
olesKonto2 = olesKonto;
```

Da får vi to referanser til det samme objektet, se nederst på figuren. Hvis vi skal ta ut kr 1000 fra Oles konto, kan vi nå skrive én av følgende to setninger:

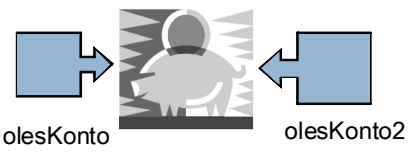
```
olesKonto.utførTransaksjon(-1000.0);     // disse to setningene har samme effekt
olesKonto2.utførTransaksjon(-1000.0);    // etter at olesKonto2 er satt lik olesKonto
```

Hvis vi utfører begge setningene, blir kr 2000 trukket fra Oles konto.

Objektet som inneholder data om den gamle kontoen til Ole, er ikke lenger tilgjengelig. Det vil etter hvert bli fjernet fra minnet.

Før: To referanser til hvert sitt kontoobjekt


Etter at setningen olesKonto2 = olesKonto; er utført:




*Det eksisterer ikke lenger noen referanse til det andre objektet, og det blir fjernet fra minnet.*

Figur 5.10 En referanse settes lik en annen referanse

## En referanse med modifikatoren final

referanser og final

Vi bruker modifikatoren final for variabler som ikke skal kunne endre verdi etter at de er initiert. Hvis en slik variabel er en referanse, er det teknisk sett kun referansen som ikke kan forandres (det vil si at den ikke kan settes til å peke til et annet objekt). Det som referansen peker til, kan forandres forutsatt at referansetypen tilbyr mutasjonsmetoder.

Eksempel 1:

Klassen String er en del av Java-API-et, og det finnes ingen metoder i denne klassen som forandrer innholdet i et objekt.

```
final String navn = "NN";
```

Her er det ikke mulig å sette navn til å peke til et nytt objekt, og det er heller ikke mulig å forandre på objektet.

Eksempel 2:

Vi lager et final-objekt av klassen Konto:

```
final Konto alltidTomKonto = new Konto(123456676756L, "NN", 0);
```

Kompilatoren protesterer hvis vi forsøker å sette alltidTomKonto til å peke til et annet objekt.

```
alltidTomKonto = new Konto(223456676756L, "NN2", 0);  // ikke ok, pga. final
```

Kompilatoren protesterer imidlertid *ikke* hvis vi prøver å sette inn penger:

```
alltidTomKonto.utførTransaksjon(1000);  // ok, selv om final er brukt
```

mutabel klasse

Begrepene mutabel og immutabel er viktige nå. Vi repeterer fra side 154:

- En mutabel klasse er en klasse som inneholder minst én mutasjonsmetode.
- En mutasjonsmetode er en metode som endrer datainnholdet i objektet.
- Et mutabelt objekt er et objekt som tilhører en mutabel klasse.

I eksemplet foran er alltidTomKonto et mutabelt objekt, utførTransaksjon() er en mutasjonsmetode, og Konto er en mutabel klasse.

En immutabel klasse er altså en klasse som ikke inneholder noen mutasjonsmetoder. Klassen String er immutabel.

Vi kan altså sette opp følgende konklusjon:

Bruk av final for mutable objekter er ingen forsikring mot at innholdet i objektet kan bli forandret. Dersom intensjonen er at datainnholdet ikke skal kunne endres, må programmereren la være å bruke mutasjonsmetoder.

Referanser kan være objektvariabler i klasser. Hvis disse er merket {readonly} i klassediagrammet, skal dette tolkes som at objektet ikke skal kunne endre verdi.

### En referanse som er null

Vi har sett flere tilfeller der det reserverte ordet null er brukt. Omsider er vi kommet så langt at vi kan forklare dette ordentlig. En referanse kan gis verdien null. Eksempel:

null

```
navn = null;
```

Referansen peker dermed ikke til noe objekt. Dette er standardverdien for objektvariabler av referansetype.

Forsøk på å kalle en metode i tilknytning til en slik referanse gir feilmeldingen NullPointerException, og programmet stopper.

NullPointerException

#### Unntaksobjekter og NullPointerException

Foran er NullPointerException nevnt. Dette er en klasse i Java-API-et.

Når en kjørefeil inntreffer, oppretter Java-tolkeren et objekt av en spesiell klasse, i dette tilfellet NullPointerException. Slike objekter kalles *unntaksobjekt* eller bare *unntak* (engelsk: exception). Vi sier at programmet *kaster unntak*. Vi kan velge å fange unntaket og behandle det på en fortrinnsvis fornuftig måte, eller vi kan kaste det videre. Foreløpig velger vi å ikke gjøre noe spesielt med disse unntaksobjektene. Det betyr at de kastes ut av main() og fører til en melding i kommandovinduet etterfulgt av programstopp.

Terminologien kan virke litt spesiell, men den er innarbeidet i mange programmeringsspråk. Unntakshåndtering behandles i sin helhet i kapittel 15.

## Oppgaver

Du skal i disse oppgavene arbeide med klassen Fag som er gitt i programliste 5.4. Du bør gjøre oppgavene i den rekkefølgen de står.

1 Her følger et lite klientprogram:

```
import static javax.swing.JOptionPane.*;
class Oppg5_3_1 {
  public static void main(String[] args) {
    String kode = showInputDialog("Fagkode: ");
    String antStudLest = showInputDialog("Antall studenter: ");
    int antStud = Integer.parseInt(antStudLest);
    Fag fag1 = new Fag(kode, antStud);
    showMessageDialog(null, "Du skrev fagkode: " + fag1.getFagkode()
    + " og ant.stud = " + fag1.getAntStud());
  }
}
```

Skriv inn programmet og kjør det. Klassen Fag henter du fra eksempelsamlingen på bokas hjemmeside. Tegn en figur som viser alle variabler og objekter som blir laget ved kjøring av programmet.

2 Sett inn setningen Fag fag2 = fag1; nederst i klientprogrammet. Juster figuren slik at den fortsatt stemmer.

3 Skriv én setning som øker antall studenter i fag1 og fag2 med 5. Kontroller med utskriftsetning.

4 Opprett et nytt objekt: final Fag fag3 = new Fag("LO191D", 40);. Hvilke konsekvenser har bruk av modifikatoren final her? Kan for eksempel antall studenter endres?

5 Deklarer og bruk variabelen Fag fag4 slik at du framprovoserer feilmeldingen NullPointerException.

**Programliste 5.4 Fag**

```
1  /**
2   * Fag.java
3   *
4   * Klassen Fag med attributtene fagkode og antall studenter.
5   * Antall studenter kan  endres.
6   */
7
8  class Fag {
9    private final String fagkode;
10   private int antStud;
11
12   public Fag(String fagkode, int antStud) {
13     this.fagkode = fagkode;
14     this.antStud = antStud;
15   }
16
17   public String getFagkode() {
18     return fagkode;
19   }
20
21   public int getAntStud() {
22     return antStud;
23   }
24
25   public void setAntStud(int antStud) {
26     this.antStud = antStud;
27   }
28 }
29
```

## 5.4 Konstruktører og initiering av objektvariabler

Objektvariabler kan gis verdi i deklarasjonen. Eksempler:

```
private final String navn = "Ola Nordmann";
private double lengde = 5;
```

startverdier for objekt-variabler

Objektvariabler merket final må initieres, enten direkte (som vist her) eller via konstruktør.

Objektvariabler som ikke er merket final, og som ikke initieres som vist her, får standard startverdier som er 0 for tallvariabler, false for logiske variabler og null for referansevariabler som for eksempel String-variabler.

konstruktør

En konstruktør brukes til å lage et objekt av klassen. Dette innebærer at det settes av plass til alle objektvariablene, og at disse initieres med sine startverdier. En konstruktør gir ofte verdier til objektvariablene. Disse verdiene vil overstyre startverdiene gitt i deklarasjonen.

La oss se på to forskjellige konstruktører for klassen Konto.

```
public Konto(long kontonr, String navn, double saldo) {
  this.kontonr = kontonr;
  this.navn = navn;
  this.saldo = saldo;
}
```

Det er den konstruktøren vi har brukt hittil. Her er det helt og holdent klientens ansvar å bestemme datainnholdet i objektet ved start.

```
public Konto(long kontonr, String navn) {
  this.kontonr = kontonr;
  this.navn = navn;
}
```

Denne konstruktøren har bare to parametere, og dersom klienten velger å bruke den, vil startsaldoen bli 0,0, som er standardverdien for tallvariabler.

En konstruktør kan kalle en annen konstruktør ved å bruke det reserverte ordet this. Konstruktøren Konto med tre parametere kan programmeres slik:

```
public Konto(long kontonr, String navn, double saldo) {
  this(kontonr, navn);  // kaller konstruktøren med to parametere
  this.saldo = saldo;
}
```

standard-konstruktør

Dersom vi ikke lager noen konstruktør i en klasse, vil det automatisk eksistere en *standardkonstruktør* (engelsk: default constructor). Denne konstruktøren svarer til en konstruktør med tom parameterliste og tom kropp:

```
public Konto() {
}
```

Standardkonstruktøren brukes slik:

```
Konto minKonto = new Konto();
```

Hvis vi skal ha en slik konstruktør i klassen Konto, må final-variablene kontonr og navn initieres direkte:

```
private final long kontonr = 0L;
private final String navn = "";
```

Standardkonstruktøren eksisterer bare hvis vi ikke lager noen konstruktører selv. Dersom vi lager egne konstruktører og ønsker at det skal eksistere en konstruktør med tom parameterliste, må vi lage også den konstruktøren selv.

signatur

*Signaturen* til en konstruktør eller metode er navnet samt antall og type parametere. Verken returtypen eller parameternavnene er del av signaturen.

Signaturen må være forskjellig for alle konstruktørene og alle metodene i en klasse.

### Oppgave

Du skal også i denne oppgaven arbeide med klassen Fag som er gitt i programliste 5.4. Du skal utvide og om nødvendig endre koden som allerede er gitt, slik at dette klientprogrammet virker:

```
public static void main(String[] args) {
   Fag fag1 = new Fag();
   System.out.println("Standardverdi for fagkode: " + fag1.getFagkode() + ".");
   System.out.println("Standardverdi for antall studenter: " + fag1.getAntStud() + ".");
   fag1.setFagkode("LO172D");
   fag1.setAntStud(47);
   Fag fag2 = new Fag("LC238D");
   fag2.setAntStud(40);
   int sum = fag1.getAntStud() + fag2.getAntStud();
   System.out.println("Totalt antall studenter er " + sum + ".");
}
```

Kjøring av programmet skal gi denne utskriften:

```
Standardverdi for fagkode: Ukjent.
Standardverdi for antall studenter: 0.
Totalt antall studenter er 87.
```

## 5.5 Metoden toString()

Følgende kodebit henter fram og skriver ut all informasjon som er lagret i objektet olesKonto:

```
long kontonr = olesKonto.getKontonr();
String navn = olesKonto.getNavn();
double saldo = olesKonto.getSaldo();
System.out.println("Oles konto: \nKontonr: " + kontonr
                                   +  ", navn: " + navn + ", saldo: " + saldo);
```

Vi trenger ikke å lagre dataene i variabler, vi kan kalle metodene direkte:

```
System.out.println("Oles konto: \nKontonr: " + olesKonto.getKontonr()
                   +  ", navn: " + olesKonto.getNavn() + ", saldo: " + olesKonto.getSaldo());
```

Vi trenger ofte å få skrevet ut datainnholdet i et objekt. Vi kan selvfølgelig gjenta setningen foran der det trengs, men det finnes en enklere måte: Vi lar objektet selv lage en passende streng. Det er en så vanlig problemstilling at det finnes et standardhode for slike metoder. For klassen Konto kan metoden se slik ut:

toString()

```
public String toString() {
  return "Kontonr.: " + kontonr + ", navn: " + navn + ", saldo: " + saldo;
}
```

I klientprogrammet kan utskriftsetningen nå se slik ut:

```
System.out.println("Oles konto: \n" + olesKonto.toString());
```

Utskriften blir slik:

```
Oles konto:
Kontonr.: 123456676756, navn: Ole Olsen, saldo: 2300.5
```

I println()-metoden er toString() underforstått dersom vi prøver å skrive ut et objekt. Det betyr at vi kan skrive følgende:

```
System.out.println(olesKonto);   // som om det hadde stått olesKonto.toString()
```

Vi kan skjøte et objekt til en streng. Da er også toString() underforstått:

```
String melding = "Oles konto: \n" + olesKonto;
```

Det eksisterer en standardutgave av metoden toString(). Den vil bli brukt dersom klassen ikke har sin egen utgave. Standardutgaven gir klassenavnet etterfulgt av @ og en tallkode. I eksemplet foran får vi i tilfelle følgende utskrift:

```
Oles konto:
Konto@87816d
```

Det er en god vane å lage toString()-metoder i alle klasser. Hensikten med metoden er som oftest å gjøre livet enklere for programmereren ved testing. Den brukes også enkelte ander steder der Java-tolkeren har behov for en tekstlig representasjon av et objekt, eksempelvis i forbindelse med grafiske brukergrensesnitt. Metoden har vanligvis ikke noen motsvarighet i den virkeligheten vi modellerer, og den vises derfor sjelden i UML klassediagram.

## Oppgave

Lag toString()-metode for klassen Fag i programliste 5.4, side 166. Prøv ut metoden.

## 5.6 Språkkjerne, klasser, forenklet utgave

Her følger forenklede syntaksbeskrivelser for klasser. Viktige tema knyttet til klasser gjennomgås også flere andre steder i boka. Syntaksregler vedrørende abstrakte klasser, subklasser og arv finnes i kapittel 14, mens enum-klasser og static behandles i kapittel 15. Indre klasser brukes i boka kun i tilknytning til GUI-programmering, og er tema i kapittel 17.3.

navnekonvensjoner for klasser

I tillegg har vi navnekonvensjoner som ikke er en del av syntaksen:

- Et klassenavn er vanligvis et substantiv i entall. Det skrives med stor forbokstav. Variabelnavn og metodenavn har liten forbokstav. Dersom navnet består av flere ord, benyttes stor forbokstav fra og med ord nummer to. Understrekingstegn og $ benyttes ikke.[13]
- Metoder som har som oppgave å hente ut eller forandre på et attributt, har standardiserte metodehoder og -navn på engelsk:

```
public void setNN(datatype NN)  // gir verdi til objektvariabelen NN
public datatype getNN()  // henter ut verdien til objektvariabelen NN
```

set-metode

Vi tar med et eksempel på en *set-metode*:

```
public void setSaldo(double saldo) {
  this.saldo = saldo;
}
```

Eventuelt kan den programmeres slik:

```
public void setSaldo(double nySaldo) {
  saldo = nySaldo;
}
```

Merk at set-metodene har returtype void. Dersom vi ønsker en annen returtype, bør vi bruke et annet metodenavn.

Dersom attributtet er av datatypen boolean, kan get- erstattes med is-:

is-metode

```
public boolean isNN()
```

Våre programmer er i norsk språkdrakt. Vi velger likevel å bruke denne navngivingen blant annet på grunn av at verktøyene vanligvis er lagt til rette for å generere disse metodene automatisk. Ettersom to av de mest brukte verktøyene (Eclipse og NetBeans) lager is-metoder for logiske variabler, bruker vi denne formen for denne typen variabler (og ikke get-).

JavaBean

Merknad: Navnestandarden for set- og get-metoder finner vi i JavaBean-spesifikasjonen [URL-JavaBean]. Egentlig er det ikke objektvariablene som bestemmer hva som skal bli get- og eventuelt set-metoder. Det er heller slik at get- og set-metodene bestemmer hva som er "properties" i JavaBean-språkbruk. Om det ligger en objektvariabel i bunnen eller dataene regnes ut eller hentes fra en database, er egentlig kli-

---

13. Unntak: Vi bruker understrekingstegn i klassenavn til løsninger av oppgaver for å angi hvilken oppgave det gjelder, eksempel Oppg5_4_2. Denne klassen viser løsningen til oppgave 2 etter kapittel 5.4.

enten uvedkommende. For klienten er dette "properties". De standardiserte navnene utnyttes blant annet i verktøy som søker etter get- og set-metoder for å identifisere et objekts "properties".

**Språkkjerne**

**Å bruke klasser**

Syntaks:

Å lage et objekt:

*klassenavn objektnavn* = new *klassenavn(argumentliste)*; // bruker en konstruktør

Metodekall:

*objektnavn.metodenavn*(*argumentliste*)

Dersom returtypen er noe annet enn void, kan vi (men må ikke) bruke metodekallet i et uttrykk.

Om argumentlister:

Argumentene i argumentlisten må stemme i type og antall med parameterlisten til en konstruktør, eventuelt en metode. Dette betyr at argumentene enten må være av samme type som parameteren eller av en type som automatisk kan omformes til denne typen (automatisk omforming, se kapittel 2.8, side 73).

Et argument kan være en konstant, et uttrykk eller en variabel. Uansett er det en *verdi* som overføres. Dersom argumentet er (verdien til) en variabel, kan, men behøver ikke, navnet på denne variabelen være det samme som navnet på parameteren.

På de følgende sidene finner du ytterligere to språkkjernebeskrivelser:

- Å deklarere en klasse
- return-setningen

**Språkkjerne**

### Å deklarere en klasse (forenklet utgave)

En klassedeklarasjon spesifiserer en referansetype. Syntaks:

```
class klassenavn {
  objektvariabler
  konstruktører
  objektmetoder
}
```

Den innbyrdes rekkefølgen av konstruktører og medlemmer inne i klassen er ikke en del av syntaksen. Hvert objekt av klassen har sitt eget sett med **objektvariabler**. Dersom en objektvariabel ikke gis startverdi, får den automatisk verdien 0 (eller false eller null, avhengig av type). Objektvariabler med modifikator final må initieres, enten direkte eller i alle konstruktørene. Syntaks for objektvariabler:

```
private final datatype navn;
private final datatype navn = startverdi;
private datatype navn;
private datatype navn = startverdi;
```

**Konstruktører** brukes til å opprette objekter av en klasse. Dersom vi ikke programmerer noen konstruktører, blir det automatisk laget en konstruktør med tom parameterliste og uten innhold (standardkonstruktøren). Alle konstruktørene i en klasse må ha forskjellig *signatur*. Signaturen består av antall og type parametere. Parameternavnene er ikke en del av signaturen. Syntaks for konstruktører:

```
public klassenavn(parameterliste) {
  setninger
}
```

En konstruktør kan kalle en annen konstruktør ved å bruke this etterfulgt av argumentliste, eksempel this(arg1, arg2);

En **objektmetode** brukes når vi skal sende en melding til et objekt. Et metodenavn kan *overloades*. Det vil si at vi kan ha flere metoder med samme navn, bare de har forskjellig *signatur* (eksempler, se klassen String, side 277, der du for eksempel finner mange metoder som heter indexOf()). Signaturen består av metodenavnet, antall og type parametere. Verken returtypen eller parameternavnene er en del av signaturen til en metode. Syntaks for objektmetoder:

```
tilgangsmodifikator returtype metodenavn(parameterliste) {
  setninger
}
```

Tilgangsmodifikatoren settes vanligvis public, men private for hjelpemetoder som kun er av interesse inne i klassen.

Metoder og variabler utgjør **medlemmene** i en klasse, og skopet er klassen. Medlemmer med offentlig tilgang kan nås utenfor klassen ved å bruke en kvalifikator, eksempel olesKonto.getSaldo().

**Språkkjerne**

**return-setningen**

Syntaks:

return *uttrykk*

eller

return

Dersom programkontrollen treffer på en return-setning, hopper den umiddelbart ut av den metoden vi er inne i, uavhengig av om det finnes flere setninger i metoden (unntaket er finally-blokker, som alltid utføres, se kapittel 15.3).

Den øverste utgaven av setningen brukes dersom metoden har annen returtype enn void. Da må *uttrykk* være av samme type som returtypen eller av en type som automatisk kan omformes til returtypen (automatisk omforming, se kapittel 2.8, side 73). En verdi returneres til den kallende metoden.

Den nederste utgaven av setningen brukes dersom returtypen er void. Denne typen return-setning brukes sjelden.

Angående switch-setningen og return: Det er ikke tillatt med break etter en return-setning i en case-blokk.

## 5.7 Nye begrep i dette kapitlet

| Begrep | Kort forklaring |
|---|---|
| accessor | Se tilgangsmetode. |
| argument | Den verdien som klienten sender sammen med en melding eller en konstruktør. Argumentet må passe i type med den tilhørende parameteren i metodehodet eller konstruktørhodet. Betegnelsen "aktuelt argument" brukes også. |
| attributt | Opplysning som et objekt har ansvaret for å vite om seg selv, en navngitt egenskap med et definert verdiområde. |
| datastruktur | En samling data lagret i primærminnet under ett navn. |
| default constructor | Se standardkonstruktør. |
| exception | Se unntaksobjekt. |
| grensesnitt | En beskrivelse av de meldingene som kan sendes til et objekt. Beskrivelsen består av metodehodene og inneholder returtype, metodenavn og parameterliste for hver enkelt metode. |

| Begrep | Kort forklaring |
|---|---|
| hode | Se klassehode, konstruktørhode, og metodehode. |
| immmutabelt objekt | Objekt som tilhører en immutabel klasse. |
| immutabel klasse | Klasse som ikke tilbyr mutasjonsmetoder. En klient kan altså ikke endre datainnholdet i et objekt av en slik klasse. |
| implementere | Å programmere klassene. Brukes også om å lage programmer generelt. |
| instans | Synonym for objekt. |
| instansiere | Synonym for den prosessen det er å lage et objekt. |
| klasse | En beskrivelse av en mengde objekter med de samme attributtene og operasjonene. I et program blir dette en logisk samling med deklarasjoner, det vil si objektvariabler og metoder. |
| klassediagram | Grafisk framstilling av en eller flere klasser og sammenhengen mellom disse, i henhold til UML. Foreløpig vil våre klassediagram kun bestå av enkeltklasser. |
| klassehode | Består av eventuell modifikator (private eller public), ordet class og klassenavnet. Eksempel: public class Person. |
| klassekropp | Hele den blokken som utgjør en klasse. |
| klient | En rolle som et objekt kan spille. Rollen innebærer at objektet ber et annet objekt (tjeneren) om en tjeneste eller stiller et spørsmål til et annet objekt. |
| klientprogram | Et program som sender meldinger til et eller flere objekter. Betegnelsen brukes ofte om metoden main(). |
| konstruktør | Brukes for å lage et objekt av en klasse. Består av konstruktørhode og konstruktørkropp. |
| konstruktørhode | Består av klassenavn og parameterliste. Eksempel: Person(String fornavn, String etternavn). |
| konstruktørkropp | Programkoden som ligger bak et konstruktørhode. Omsluttes av {}. |
| kropp | Se klassekropp, konstruktørkropp, og metodekropp. |
| kvalifikator | Alle delene i et kvalifisert navn unntatt siste del. |
| kvalifisere et navn | Nødvendig når vi bruker navnet utenfor skopet. Kvalifikatoren vil være pakkenavn (f.eks. javax.swing), klassenavn eller navnet på et objekt. |

| Begrep | Kort forklaring |
|---|---|
| medlem | Alle konstanter og variabler deklarert utenfor metodene i en klasse samt metodene selv, er medlemmer i klassen. Konstruktører er ikke medlemmer. |
| melding | En instruks eller et spørsmål som klientobjektet sender til tjenerobjektet. Programmeres ved at en metode kalles, eksempel: minKonto.utførTransaksjon(3000); |
| metode | Deklarasjoner av operasjoner på programkodenivå. En metode består av metodehode og metodekropp (se disse). |
| metodehode | Består av metodens returtype (eventuelt void), navn og parameterliste. Eksempel: String getNavn(). |
| metodekall | Det at en klient sender en melding til et tjenerobjekt. Data sendes med som (aktuelle) argumenter. |
| metodekropp | Programkoden som ligger bak et metodehode. Omsluttes av {}. |
| mutabel klasse | Klasse som tilbyr mutasjonsmetoder, dvs. at en klient kan forandre på datainnholdet i et objekt av en slik klasse. |
| mutabelt objekt | Objekt som tilhører en mutabel klasse. |
| mutasjonsmetode | Metode som endrer verdien til et eller flere attributter. |
| mutator | Se mutasjonsmetode. |
| new | Reservert ord for å lage et objekt av en klasse. |
| null | En variabel av referansetype kan gis verdien null. Det betyr at den ikke refererer til noe objekt. |
| objekt | De fleste objektene i et program er modeller av virkelige objekter. Modellobjekter har mye kunnskap om seg selv (attributter) og om hvordan det skal løse oppgaver (operasjoner), uavhengig av om det virkelige objektet er levende, dødt eller abstrakt. Eksempler på objekter i programmer som ikke er modeller av virkelige objekter, er brukergrensesnittobjekter. |
| objektmetode | Representerer en operasjon. En objektmetode må kalles på vegne av et objekt, dette i motsetning til klassemetoder (se kapittel 9.1). Kalles ofte bare "metode" dersom det ikke er fare for misforståelser. |
| objektvariabel | Representerer vanligvis et attributt. Deklareres utenfor alle metodene i en klasse. Hvert objekt av klassen har sitt eget sett med objektvariabler, dette i motsetning til klassevariabler (se kapittel 15.5). |
| operasjon | De instruksjonene som et tjenerobjekt utfører som svar på en melding. |

| Begrep | Kort forklaring |
|---|---|
| overloade navn | I en klasse kan metoder og konstruktører med samme navn holdes fra hverandre ved at de har forskjellig signatur. Vi sier at navnene er overloadet. |
| parameter | Beskriver en verdi som en klient må sende sammen med en melding eller en konstruktør. Beskrivelsen omfatter datatype og et beskrivende navn. Betegnelsen "formell parameter" brukes også. |
| peker | Se referanse. |
| private | Modifikator som forteller at for eksempel en objektvariabel ikke er tilgjengelig utenfor klassen der det er deklarert. |
| public | Modifikator som forteller at for eksempel en metode er tilgjengelig utenfor klassen der den er deklarert. |
| referanse | En variabel av referansetype. Innholdet i variabelen kan betraktes som adressen til et objekt av den gitte typen. |
| referansetype | En datatype som er beskrevet ved hjelp av en klassedeklarasjon. |
| return-setningen | Setning som medfører umiddelbart uthopp fra en metode. |
| returtype | Datatypen til returverdien fra en metode. |
| returverdi | Den verdien som en metode sender tilbake til klienten. Det vil være tjenerens svar på en forespørsel fra klienten. |
| signatur | Navnet på konstruktør eller metode samt antall og type parametere. Verken returtypen eller parameternavnene er en del av signaturen. |
| standard-konstruktør | En konstruktør som blir laget automatisk dersom vi ikke lager konstruktører selv. Den svarer til en konstruktør med tom parameterliste og tom kropp. |
| this | Reservert ord som betegner en referanse til det objektet vi holder på med akkurat nå. Referansen kan brukes fra objektmetoder. |
| tilgangs-metode | Metode som henter ut en attributtverdi eller et beregnet resultat. Metoden endrer ikke på attributtverdiene. |
| tjener | En rolle som et objekt kan spille. Rollen innebærer at objektet besvarer et spørsmål eller utfører en oppgave etter instruks fra et annet objekt (klienten). |
| unntak | Se unntaksobjekt. |
| unntaksobjekt | Et objekt som beskriver en feilsituasjon. Objektet tilhører en såkalt "exception"-klasse. Mer om dette i kapittel 15. |
| void | Reservert ord som forteller at metoden ikke returnerer noen verdi. |

## 5.8 Repetisjonsoppgaver

1 Hva er sammenhengen mellom begrepene objekt og klasse?

2 Et objekt i et program er ofte en modell av et objekt i virkeligheten. Hvilke vesensforskjeller har vi gjerne mellom oppførselen til de to objektene?

3 Hvorfor kaller vi gjerne main()-metoden for klientprogram?

4 Et tjenerobjekt kan utføre visse tjenester for et klientobjekt. På hvilken måte vises disse tjenestene i et klassediagram? Enn i programkoden?

5 Oppbygningen av konstruktører og metoder har mange fellestrekk, men også noen forskjeller. Beskriv fellestrekk og forskjeller.

6 Hva bruker vi konstruktører til? Enn metoder?

7 I hvilken sammenheng bruker vi returverdier?

8 Hva betyr {readonly}, + og - i et klassediagram? Hvordan vises dette i programkoden?

9 Hva er forskjellen mellom parameter og argument? Vis gjerne med et eksempel.

10 Er objekt og referanse det samme? Hvis ikke, hva er forskjellen?

11 Hva skjer hvis vi setter en referanse lik en annen? Vis ved et eksempel og en figur.

12 Hvilke konsekvenser har det å sette final foran en objektvariabel?

13 Anta at vi lager en klasse uten å programmere konstruktører. Er det mulig å lage objekter av denne klassen? Begrunn svaret.

14 Hva mener vi når vi snakker om signaturen til en metode?

## 5.9 Programmeringsoppgaver

### Oppgave 1

Lag en klasse Prosjekt. Prosjekttittel, navn på ansvarlig person samt budsjett (kun ett tall) skal legges inn ved prosjektstart. Prosjektet skal også holde rede på egen økonomi. Klassen skal ha metoder for å hente ut hver enkelt av opplysningene samt en metode som registrerer at et pengebeløp er påløpt prosjektet. Klassen skal også ha en toString()-metode.

Klientprogrammet skal lage et prosjektobjekt og la brukeren skrive inn flere påløpte beløp. For hvert beløp skal prosjektobjektet oppdateres og all informasjon om prosjektet skrives ut.

### Oppgave 2

Lag en klasse Valuta med minst én konstruktør. Klassen skal ha metoder for å regne fra og til norske kroner.

Lag et klientprogram som oppretter minst tre objekter som representerer forskjellige valutaer. Brukeren skal få tilbud om å regne om flere ulike beløp i de forskjellige valutaene til norske kroner.

Programmet må altså presentene en meny for brukeren. Den kan for eksempel se slik ut:

```
Velg valuta:
1: dollar
2: euro
3: svenske kroner
4: avslutt
```

Brukeren skriver inn ett av tallene 1, 2 eller 3 (eller 4 for avslutt). Dette skal styre programflyten slik at riktig valutaobjekt blir brukt.

## Oppgave 3

Du skal programmere terningspillet 100 med to spillere, A og B. Reglene er som følger:

A og B kaster terningen annenhver gang. Antall øyne på terningen er antall poeng oppnådd i denne runden. Poengene summeres fortløpende. Dersom en spiller kaster 1, settes summen tilbake til 0. Den som først passerer 100 poeng, har vunnet spillet.

Lag en klasse som simulerer en spiller. Den kan ha sumPoeng som objektvariabel og metoder som heter getSumPoeng(), kastTerningen() og erFerdig().

For å simulere terningen skal du bruke klassen java.util.Random fra Java-API-et (se side 238). Denne klassen brukes til å lage rekker med tilfeldige tall. La et objekt av klassen være objektvariabel:

```
java.util.Random terning = new java.util.Random();  // vi lager en tilfeldigtallgenerator
```

I metoden kastTerningen() henter du et tilfeldig tall i intervallet [0, 5] på følgende måte:

```
int terningkast = terning.nextInt(6);
```

Legg til 1 for å få et gyldig terningkast.

Lag et klientprogram som oppretter to objekter av klassen, ett for hver av spillerne. Lag en løkke som kjører inntil en av spillerne har vunnet. I hvert gjennomløp av løkken skal hver spiller kaste terningen én gang. Skriv ut rundenummer og poengsummene til hver av spillerne i hvert gjennomløp. Det er sannsynligvis mest praktisk å skrive til kommandovinduet her.

En raffinering av spillet kan være å kreve at man skal komme akkurat til 100. Dersom denne grensen er passert, skal neste kast trekkes fra poengsummen. Hvis man nå havner akkurat på 100, er man ferdig. Ellers legges neste kast til, og slik holder man på og legger til og trekker fra inntil man kommer akkurat til 100.