# Litt Java for deg som kan Simula

Stein Krogdahl

Januar 1999

## Innhold

# 1 Innledning

Dette notat er laget for å lette overgangen fra Simula til Java som undervisninsspråk ved Institutt for informatikk. Dette notatet tar derfor i noen grad utgangspunkt i at leseren har vært borte i Simula, men dette er nok slett ikke noen forutsetning for å få utbytte av notatet. Leseren bør nok imidlertid ha noe kunnskap om programmering, og gjerne også om objektorientering, på forhånd.

Dette notat er laget uten detaljert forhåndskunnskap om Java, og med en nokså knapp frist. Det vil derfor sikkert inneholde en del tradisjonelle trykkfeil, og, på tross av intens lesing og uttesting av det aller meste, kan det sikkert også være påstander om Java som er ufullstendige eller direkte gale. Forfatteren mottar gjerne kommentarer på alle plan om notatet.

Det som kanskje slår Simula-kjennere mest når de studerer Java er hvor likt det er Simula, om man ser bort fra overflatiske syntaktiske forskjeller. Ser man litt på historien er imidlertid ikke det så underlig, for den kan MEGET kort beskrives slik:

```
C++ = C + Simula
Java = C++ - C
```

Vi kan altså si at Java er "Simula i C-syntaks". Java er derfor rimelig lett å lære for Simula-kjennere.

Et par ting om terminologi: Vi har i dette notatet brukt ordene 'setning', 'imperativ' og 'deklarasjon' slik at 'setning' dekker både 'imperativ' (altså "utførbar setning") og 'deklarasjon'. Skillet mellom deklarasjoner og imperativer er noe mer uklart i Java enn i Simula (de kan ofte blandes fritt innen en blokk), og vi bruker derfor ofte begrepet 'setning' der det ikke eksplisitt er viktig å skille.

I Java finnes klasser og subklasser omtrent som i Simula. Som i Simula kan det derfor være tvil om hva som menes når vi snakker om f.eks. "alle subklassene" til en klasse K. Mener man bare de "direkte" subklassene eller inkluderer man også subklasser av subklasser, og eventuelt også klassen K selv? Vi skal bruke følgende terminologi, som hvertfall også er brukt i noen Java-bøker:

Det som i Simula skrives 'A class B' skrives i Java 'class B extends A'. I dette tilfelle skal vi si at B er en "direkte subklasse" av A og at A er en "direkte superklasse" av B. Når vi bare sier at C er en "subklasse" av A skal vi mene at *enten er C og A samme klassen*, eller C er en direkte subklasse av C1, som igjen er en direkte subklasse av C2, ..., som til slutt er en direkte subklasse av A. Om vi vil ekskludere at klassene kan være like vil vi snakke om "ekte subklasse". Tilsvarende gjelder når vi sier "superklasse" eller "ekte superklasse".

## 1.1 Oversikt over notatet

I kapitlene som følger skal vi ta for oss de viktigste sidene Java, og inndelingen blir som følger:

I **kapittel 2** gir vi en oversikt over hvordan et Java-program fremstår som tekst på en fil, og hvordan dette skiller seg fra et Simula-program. Vi gir her også et eksempel på et helt Java-program.

Dernest gir vi, i **kapittel 3**, en oversikt over hvordan de forskjellige konstruksjoner ser ut i Java, og ser litt på hva som mangler og kommer i tillegg i forhold til i Simula.

Videre i **kapittel 4** skal vi se litt på Javas datatyper og på deklarasjoner av variable. I den forbindelse ser vi også litt på beregninger og tilordning. Generelt er her forskjellene fra Simula små, men vi skal peke på en del detaljer som man bør være klar over.

Temaet for **kapittel 5** er arrayer og behandling av tekster. En array er i Java eksplisitt å betrakte som et objekt, og må skapes med en new-operasjon. Bak kulissene er det også slik i Simula, men for programmereren blir det her en del nye ting ved overgang fra Simula til Java.

I **kapittel 6** ser vi på begrepene klasse og subklasse, og på variabel-deklarasjoner i disse. Svært mye av mekanismene rundt subklasser er imidlertid koblet til "metode"-deklarasjoner, og dette taes i neste kapittel.

I **kapittel 7** ser vi så på prosedyrer, eller som de kalles i Java: "metoder". Disse vil syntaktisk alltid ligge lokalt inne i klasse-deklarasjoner. Vi ser her på hvordan parametere overføres og hvordan man returnerer (og eventuelt leverer et resultat) fra en slik metode. Videre er det en del detaljer i koblinga mellom forskjellige typer metoder og klasse/subklasser.

Videre går vi, i **kapittel 8**, inn på en mekanisme som det ikke finnes noe tilsvarende til i Simula, nemlig såkalt "unntakshåndtering" ("exception handling"). Dette er en mekanisme for fange opp "spesielle" hendelser under utførelsen, og å kunnne behandle dem på en kontrollert måte uten at de forstyrrer hovedstrukturen i programmet unødig.

I **kapittel 9** går vi så inn på hvordan man kan la et Java-program være spredt på flere filer, og hvordan man kan la en eller flere filer til sammen danne en "pakke". Her gir vi også en oversikt over hvordan man kan styre synligheten av navn mellom de forskjellige bitene av programmet.

I **kapittel 10** går så litt inn på kompilering og kjøring av Java-programmer, og hva slags filer som oppstår under veis etc.

Endelig går vi i **kapittel 11** litt inn på en spesiell type "ekstra abstrakte" klasser, som kalles "interfacer". Disse kan sees som en slags "briller" som vi kan se på objekter gjennom, og som gjør at vi bare ser de delene av objektet som denne brillen tillater. Man kan se på samme objektet med forskjellige briller.

## 1.2 Ting i Java som ikke behandles i dette notatet

Dette notatet er bare ment å skulle få på plass de mest basale delene av Java. Det er derfor flere sider av språket og dets omgivelser som ikke behandles, og de viktigste av disse er følgende:

**Lager-administrasjon i Java:** I likhet med Simula har Java en lageradministrasjon med automatisk gjenbruk av plass under utførelsen, altså såkalt automatisk søppeltømming ("garbage collection"). Man slipper defor å tenke på at man eksplisitt må frigi de objektene man ikke har bruk for lenger. Når man ikke kan få tak i dem mer blir plassen automatisk frigtitt.

**Utskrift og innlesning:** Vi viser i programmer noen få eksempler på utskrift. Utskrift og innlesning gjøres generelt av klasser/metoder som ligger på et eget bibliotek, og hverken dette eller andre standard-biblioteker gå vi inn på i dette notatet. Det finnes et utall av ferdiglagede bibliotek for Java.

**Grafikk:** Mange av de ferdige bibliotekene har å gjøre med produksjon av grafikk på skjermen. Disse er en viktig del av Java-systemet, men det fører altså for langt å gå inn på det her.

**Tråder:** I Java finnes ikke noe som direkte tilsvarer det (lite kjente) apparat med ko-rutiner i Simula. Derimot har man et system der flere deler av programmet kan utføres i parallell, og hver slik del kalles da en tråd ("thread").

**Indre klasser:** I siste versjon av Java (versjon 1.2, også kalt "Java 2") er det til en viss grad lov å deklarere klasser inne i klasser, men det skal vi ikke se på i dette notatet.

**Applets:** I stedet for å skrive et Java-program som et hoved-program kan det skrives slik at det lett kan hentes over nettet av en nett-leser, og utføres i denne (gjerne med levende grafikk som resultat). Slike programmer kalles "applets", og de beskrives mange steder, men ikke her.

# 2 Hoved-strukturen i et Java-program

Java-språket er i stor grad orientert mot at forskjellige deler av et program ligger på forskjellige filer. I første omgang skal vi imidlertid bare se på det tilfellet at hele programmet (bortsett fra nødvendige biblioteks-rutiner til I/O etc.) ligger sammenhengende på én tekst-fil.

Det som i Simula heter en prosedyre heter altså i Java (og i mange andre OO-språk) en "metode". Vi skal i det følgende holde oss til denne terminologien (men fremdeles bruke begrepet prosedyre når vi snakker om ting i Simula).

## 2.1 Blokkstruktur i Simula og Java

Et Simula-program kan som kjent ha blokk-struktur der alle typer blokker og deklarasjoner (klasser, prosedyrer, variable, ...) kan forkomme inni hverandre, nokså uten begrensninger. I Java er det derimot klare restriksjoner på dette, og grovt sett gjelder følgende: Ytterst er det bare klasse-deklarasjoner, inni klassene er det bare metode- og variabel-deklarasjoner, mens det inni metodene og videre innover bare kan være variabel-deklarasjoner. Det er også mulig å erstatte noen av klasse-deklarasjonene med såkalte "interface"-deklarasjoner, og dette ser vi litt på i et senere kapittel.

Et Java-program har altså ikke noen ytre "ramme", slik som den ytterste 'begin' - 'end' rundt et program i Simula. Det er det at programteksten i Java ligger på en felles fil som gjør det til ett program. Litt mer i detalj ser et Java-program ut som følger:

Programmet starter med et antall import-setninger, som angir hvilke ferdigkompilerte klasser (gjerne biblioteks-klasser) som skal brukes. Resten av programmet er en sekvens av klasse-deklarasjoner. Én av disse må ha samme navn som fila programmet ligger på (bortsett fra endelsen ".java" på filnavnet). Denne klassen er å betrakte som programmets "hoved-klasse", og navnet på denne kan vi også betrakte som navnet på dette Java-programmet.

Inni hver klasse kan det bare være metode-deklarasjoner og variabel-deklarasjoner. Alle imperativ-setninger blir derved liggende inni metoder igjen.

I Simula kan man i selve klassen angi imperativer som blir utført når man skaper et objekt av klassen. I Java kan man få til noe tilsvarende på en litt annen måte, nemlig ved inne i klassen å deklarere en metode med samme navn som klassen. En slik metode kalles en "konstruktør", og den vil automatisk bli kalt når man skaper et objekt av klassen.

Inne i metodene kan vi bare ha variabel-deklarasjoner, og inne i setningene i metodene kan det igjen være "blokker" (dannet ved '{' og '}' i stedet for Simulas 'begin' og 'end') med nye variabeldeklarasjoner, og disse deklarasjonene behøver ikke en gang å stå i starten av den blokken de tilhører. Vi skal i det videre kalle en '{' ... '}'-setning for en *blokk*, enten det er deklarasjoner i den eller ikke.

## 2.2 Navn og synlighet (skop-regler)

Når det gjelder navn på klassene og navn på de lokale deklarasjone i klassene så behøver disse ikke være deklarert tekstlig før det stedet man refererer til dem. Rekkeføgen av deklarasjonene spiller altså liten rolle, akkurat som i Simula.

Merk også at man i Java i en del tilfeller ikke får lov å gjenbruke et navn (f.eks. på en variabel) i en indre blokk, når det allerede er brukt i ytre blokk. Dette forbudet gjelder grovt sett for navn brukt inne i samme metode (men i "sidestilte" blokker kan man godt bruke samme navn).

En annen litt spesiell ting er følgende: Inne i metodene (eventuelt i indre blokker) kan man deklarere variable inne mellom vanlige utførbare imperativer. Disse deklarasjonen vil da ikke være refererbare før (tekstlig) etter det sted de er deklarart, og de vil alltid slutte å eksistere ved slutten av den nærmest omsluttende blokk.

I tillegg til disse skop-reglene for synlighet kan synligheten eventuelt restrikteres ytterligere ved å bruke nøkkelordene 'public', 'protected' eller 'private'. Dette forklares lenger ned.

Det finnes ikke noe spesielt nøkkelord som angir at noe er en metode, så man gjenkjenner dem ved at det står en (eventuelt tom) parentes med formelle parametere bak metode-navnet.

Metoder kan levere et resultat, og man skal da, som i Simula, foran metode-deklarasjonen angi typen på det metoden skal levere. Om den ikke skal levere noe resultat bruker man her i stedet nøkkelordet 'void', altså "ingen type".

## 2.3 "Globale" deklarasjoner

Siden det bare er klasser som kan deklarares på ytterste nivå, har et Java-program ikke globale variable og globale metoder/prosedyrer i samme forstand som i Simula. Det er imidlertid likevel mulig å angi en variabel-deklarasjon som resulterer i én og bare én variabel som eksisterer under hele programmets utførelse. Slike må imidlertid alltid være knyttet til en klasse, og man oppnår dette ved å sette nøkkelordet 'static' foran en (ellers vanlig) variabel-deklarasjon inne i klassen.

En slik "statisk variabel" er dermed felles for alle objektene i klassen, og fra alle disse objektene kan man referere til en slik variable direkte ved navn. Om man vil snakke om den fra en annen klasse må man bruke konstruksjonen '<klasse-navn>.<variabel-navn>'.

Man kan også bruke nøkkelordet 'static' foran metode-deklarasjoner i klasser, og får dermed noe som tilsvarer Simulas globale prosedyrer. Disse er altså felles for alle klassens objekter, og kan aksessers på samme måte som statiske variable. Inne fra slike statiske metoder kan man bare refere (direkte) til statiske variable og statiske metoder i egen klasse (og i super-klasser).

## 2.4 Oppstart av utførelsen

Endelig skal vi se på hvordan man starter opp utførelsen av et Java-program. Dette styres ved at hoved-klassen *må* ha en metode ved navn "main", og denne *må* være deklarart 'public', 'static' og 'void', og den *må* ha en parameter der den kan motta et antall tekst-strenger fra den som starter opp programmet. Programutførelsen starter altså ved at operativsystemet "kaller" denne metoden, som f.eks. kan se slik ut:

```
public static void  main (String[] args)
{ .... }
```

## 2.5 Kommentarer

Kommentarer kan i Java angis på to måter:

```
1: Om man skriver '//' blir dette og resten av linja ansett som
   kommentar.

2: Om man skriver '/*' blir dette og fram til og med første forekomst
    av '*/' (gjerne på en senere linje) ansett som kommentar.  Om man
   skriver:

              .... /* bbbb /* cccc */ dddd */ ....

   så vil man IKKE få noen parentetisk virkning. Teksten "dddd */" vil
   altså bli forsøkt tolket som vanlig programtekst
```

## 2.6 Et eksempel-program i Java

Som eksempel på et Java-program kan man studere følgende. Merk altså at '{' og '}' nokså konsekvent brukes tilsvarende 'begin' og 'end' i Simula.

```
import java.io.*;  // Bibliotheksklassene som gjør input/output

public class LiteProg
{
  public static void  main (String[] args) // Denne "kalles" fra op.sys.
  {  int i = 1;                          // Deklarasjon med initialisering

     System.out.println("Velkommen til vårt Bank-system");

     Konto a= new Konto("Per");          // Deklarasjon med initialisering
     a.settinn(27182);                   // Imperativ: Kall på metode

     LonnsKonto b;                       // Deklarasjon etter imperativ
     b= new LonnsKonto("Pål", 19400);    // Vanlig peker-tilordning
     b.settinn(31415);

     a.oppgjor();  // Får versjonen i Konto
     b.oppgjor();  // Får versjonen i LonnsKonto

     System.out.println("Takk, dette var alt for i dag");
  }
}

class Konto
{  String navn;
   int sum;

   Konto (String n)  // Konstruktør
   { navn= n; sum= 0;                    // Vanlig verdi-tilordning
     System.out.println("Skaper konto for: " + navn );
   }
```

```
   void settinn(int s)                  // Vanlig metode
   {sum= sum + s;}

   void oppgjor()                       // Vanlig metode
   { System.out.println("  Konto-oppgjør av vanlig konto...");
     // Som DET skal gjøres ....
   }
}

class LonnsKonto extends Konto          // LonnsKonto er subklasse av Konto
{  int lonn;

   LonnsKonto (String n, int l)         // Konstruktør
   { super(n);                          // Kall på superklassens konstruktør
     lonn= l;
     System.out.println("   som er lønns-konto, med lønn: " + lonn);
   }

   void oppgjor()                       // Ny variant for lønnskonto
   { System.out.println("  Konto-oppgjør av lønns-konto...");
     // Som DET skal gjøres ....
   }
}
```

### 2.7 Kompilering og utførelse av programmet

Kompilering og kjøring av et Java-program gjøres vanligvis med følgende kommandoer.

```
- Legg programmet på en fil med navn LiteProg.java
- Gi kommandoen >javac LiteProg.java   (dermed gjøres kompileringen)
- Gi kommandoen >java LiteProg (dermed startes det, om kompilering gikk OK)
```

## 3 Noen basale konstruksjoner i Java

For å skape litt oversikt skal vi her gå gjennom en del konstruksjoner i Java, og i noen grad kommentere forskjellene til Simula. Flere av disse konstruksjonene kommer vi nærmere tilbake til siden.

### 3.1 Bruk av '{', '}' og semikolon

Grovt sett bruker man altså '{' og '}' som man bruker 'begin' og 'end' i Simula, og vi skal altså i det følgende kalle en '{' ... '}'-setning for en *blokk* enten den har deklarasjoner eller ikke. Man bruker i Java semikolon som avslutningstegn for de fleste konstruksjoner. Det er imidlertid en del detaljer som er forskjellige, og som man må passe på. Forskjellene kan sammenfattes som følger:

1. Både imperativer og deklarasjoner *må* generelt avsluttes med semikolon, også om de er siste setningen foran en '}'. Ja, semikolonet er faktisk å anse som en del av deklarasjonen eller imperativen. Det er imidlertid ett viktig unntak fra dette, og det er at en blokk, altså en '{'

... '}'-setning, implisitt inkluderer et semikolon, og er dermed syntaktisk å anse som likeverdig med en enkel setning med semikolon etter. Dette er den samme filosofien som i C og C++.

2. Det er lov å ha både tomme deklarasjoner og tomme imperativer. Dermed kan man i stor grad putte på så mange ekstra semikoloner man lyster. (I Simula er det lov med tomme imperativer, men ikke med tomme deklarasjoner). Merk imidlertid at dette ikke gjelder foran 'else' i en if-setning. Der må man være meget nøye med hvordan/om man setter semikolon.

3. Kroppen på en klasse og en metode *må* være en blokk. Og i og med at kroppen dermed implisitt inkluderer et semikolon behøver man ikke å avslutte en slik deklarasjon med semikolon. Man kan altså skrive en sekvens av klasse- eller metode-deklarasjoner uten semikolon mellom. Man kan imidlertid gjerne putte på semikolon, da blir disse bare ansett som avslutning av tomme deklarasjoner.

Vi gir til slutt noen eksempler på gale og riktige konstruksjoner, og vi lar her S stå for en enkel setning, f.eks en tilordningssetning. Tilsvarende står B for et passelig boolsk uttrykk.

```
1. Riktig setnings-konstruksjon:
    { S; S; S;} S; ;

2. Gal setnings-konstruksjon:
    { S; S; S} S; S;

3. Også riktig, det er lagt inn en tom setning:
    { S; S; S;}; S; S;

4. Riktig sekvens av klasse-deklarasjoner:
    class C1{ ... }
    class C2{ ... }
    class C3{ ... }


4. Også riktig sekvens av klasse-deklarasjoner:
    class C1{ ... };
    class C2{ ... };;;
    class C3{ ... };

5. Riktige if-setninger (mer om if-setninger senere):
    if (B) S; else S;
    if (B) {S; S; S;;} else S;

5. Gale if-setninger:
    if (B) S else S;
    if (B) {S; S; S;}; else S;
```

## 3.2 Klasse- og metode-deklarasjoner

Vi kommer i senere kapitler tilbake til klasser og metoder, og vi henviser foreløpig til det tidligere eksempelprogrammet for å få et første inntrykk av syntaksen etc. Merk altså at kroppen både i en klasse og i en metode *må* være en blokk.

## 3.3 Variabel-deklarasjoner

En enkel variabel-deklarasjon av en basal-type er svært lik den i Simula:

```
<basal-type> a, b, c;
```

Dette betyr altså at tre enkle variable med den angitte typen blir opprettet.
Det som i Simula ser ut som 'ref(C) x, y, z;', skrives i Java rett og slett slik:

```
C x, y, z;
```

Det som i Simula heter 'none' heter i Java 'null'. Mer om deklarasjoner, initialiseringer, konstanter etc. kommer senere.

## 3.4 Arrayer

Arrayer organiseres litt annerledes i Java enn i Simula, og vi kommer tilbake til det. Vi kan her bare fastslå at følgende deklarasjoner i Simula og Java er ekvivalente:

```
Simula:         integer array A(0:29);
Java:           int[] A = new int[30];
(Også riktig:   int A[] = new int[30];)
```

## 3.5 If-setning

En if-setning skrives i Java gjerne som følger (uten og med else-gren):

```
if (Betingelse) { Setninger }
if (Betingelse) { Setninger } else { Setninger }
```

Merk her at man kan erstatte '{ Setninger }' med f.eks. en enkel tilordnings-setning, men om vi gjør det foran 'else' må vi ha nøyaktig ett semikolon etter setningen.
Merk også at følgende konstruksjon i utgangspunktet er flertydig ("hvilken 'if' hører sammen med 'else'?"):

```
if (Betingelse) if (Betingelse) { Setninger } else { Setninger }
```

Denne flertydigheten blir pr. definisjon løst slik at hver 'else' kobles til nærmeste (ledige) 'if'. Setningen blir altså tolket slik:

```
if (Betingelse) { if (Betingelse) { Setninger } else { Setninger } }
```

## 3.6 While-setning

Denne virker helt rett fram, og ser ut som følger:

```
while (Betingelse) { Setninger }
```

Her kan man selvfølgelig også erstatte '{ Setninger }' med f.eks. en enkel tilordnings-setning.

## 3.7 For-setning

For-setningen er litt annerledes enn i Simula, men er svært lik den i C++.

```
for (Initialisering; Betingelse; Steg) { Setninger }
```

Dette betyr pr definisjon det samme som:

```
{ Initialisering; while ( Betingelse ) { Setninger; Steg; } }
```

Merk her at det er satt på en setnings-parentes rundt hele konstruksjonen. Det er fordi man tillater at Initialiseringen er en deklarasjon (som da må ha initialisering), og denne eksiterer pr. definisjon bare inne i for-setningen.
Noen typiske for-setninger kan være:

```
for (i= 1; i <= 100; i= i+1) { Noen passelige setninger }
for (int i= 1; i <= 100; i= i+1) { Noen passelige setninger }
```

Det kan i Steget være behagelig å bruke de spesielle tilordnings-operatorene som blir diskutert lenger ned, f.eks. slik:

```
for (i= 1; i <= 100; i+= 2) { Noen passelige setninger }
for (i= 1; i <= 100; i++) { Noen passelige setninger }
```

Noe som er litt spesielt, men som kan være greit å kjenne til om man ser det, er følgende: Det er i for-setningen litt større frihet enn antydet over, i og med at man kan initialisere (og gjerne deklarare) *flere variable av samme type* i initialiserings-delen, og da med komma mellom. Tilsvarende kan man ha flere tilordninger i Steg-delen, også her med komma mellom. Eksempler:

```
for (int i= 1, j= 3; i+j <= 100; i+= 2, j++) { Noen passelige setninger }
int ii, jj;
for (ii= 1, jj= 3; ii<= 100; ii+= 2, jj++) { Noen passelige setninger }
```

## 3.8 Switch-setning

I stedet for en lang if-then-else kan man i visse tilfelle erstatte det med en switch-setning, som har (eller vanligvis brukes på) følgende form:

```
switch ( Seleksjon-uttrykk ) {
  case konstant1: Setninger; break;
  case konstant2: Setninger; break;
  ...
  case konstantN: Setninger; break;
  default:        Setninger;
}
```

Her må seleksjons-uttrykket enten være et heltallsuttrykk eller et char-uttrykk, og alle konstantene og seleksjonsuttrykket må ha samme type. Dersom man ikke har 'break' på slutten av et case-alternativ vil kontrollen bare "falle gjennom" til neste alternativ (mer om break etc. under). Det er (dessverre) bare lov å angi enkeltkonstanter (ikke f.eks. et intervall) etter hver 'case', men ved å droppe 'Setninger; break;' bak et alternativ får man slått dette sammen med neste, og man kan derved hvertfall lett få samme aksjon for mange konstanter.

### 3.9 Inspect, goto, break, continue m.m.

Det finnes i Java ikke noe som tilsvarer Simulas inspect-setning (eller Pascals with-setning). Vil man utenfra og inn i et objekt *må* man altså bruke dot-notasjon.

Det er heller ikke noe i Java som direkte tilsvarer Simulas goto-setning, men det er derimot et antall mekanismer som er ment å erstatte "ryddig" bruk av 'goto'. Disse er følgende:

**'break'-setning:** Om man rett og slett sier 'break' vil man gå ut av nærmeste omsluttende while-, for-, eller switch-setning, og fortsette på neste setning. En 'break' er ulovlig om det ikke er noen slik omsluttende setning.

Man kan ellers sette en label på nesten en vilkårlig setning (gjøres som i Simula, med et navn med kolon etter), og inne i denne setningen kan man si 'break <label>;' for å hoppe ut og fortsette bak setningen med labelen. Dette kan brukes til å hoppe flere nivåer ut.

**'continue'-setning:** Om man rett og slett sier 'continue' vil man gå til neste iterasjon av den nærmeste omsluttende while- eller for-setning (den går altså til slutten av kroppen, og gjør normale forberedelser til en eventuell ny runde). Denne setningen er ulovlig om det ikke er noen slik omsluttende while- eller for-setning.

Også for 'continue' kan man bruke en label, men labelen må da stå på en omsluttende while- eller for-setning. Man går da tilsvarende til slutten av den aktuelle setnings body, og starter som normalt på ny runde i denne.

**'return'-setning:** Denne brukes inne i metoder, og vil rett og slett avslutte utførelsen av metoden og returnere kontrollen til kallstedet. Dersom metoden skal returnere et resultat *må* man bruke denne måten av avslutte metoden på, og bak 'return' *må* man da angi et uttrykk som gir det resultatet som skal returneres.

**Unntaks-håndtering:** Det finnes i Java en mekanisme for å si fra at det er skjedd noe spesielt (man "kaster et unntak" med en throw-setning), og at man ønsker at kontrollen skal ledes til et sted der man har angitt hvordan slike situasjoner skal behandlers. Denne mekanismen kommer vi tilbake til i et senere kapittel.

## 4 Litt om typer, uttrykk, beregninger og tilordning

### 4.1 Aritmetiske typer

Java har følgende 6 aritmetiske typer, og vi skal si at den rekkefølgen de her er satt opp i går fra "svakere" til "sterkere" aritmetiske typer:

```
Heltall:
   byte:   en byte, med fortegn
   short:  to byter, med fortegn
   int:    fire byter, med fortegn
   long:   åtte byter, med fortegn

Reelle tall:
   float:  fire byter
   double: åtte byter
```

For aritmetsike operander gjelder:

```
        Simula                        Java
-----------------------------------------------------
      +, -, *, /                    Som i Simula
      // (heltallsdivisjon)         / (med begge operandene heltall)
      mod(I,J)                      I % J  (resten, ved I/J)
```

For de aritmetiske typene finnes i Java noen spesielle tilordnings-operatorer som vi viser med eksempler (der I er en tallvariabel)

```
++I // Variablen økes med 1, og verdien av det hele er den nye I-verdien
I++ // Variablen økes med 1, og verdien av det hele er den gamle I-verdien
--I // Variablen minkes med 1, og verdien av det hele er den nye I-verdien
I-- // Variablen minkes med 1, og verdien av det hele er den gamle I-verdien
I+= a;  // Betyr det samme som I= I+a;
I-= a;  // Betyr det samme som I= I-a;
I*= a;  // Betyr det samme som I= I*a;
I/= a;  // Betyr det samme som I= I/a;
```

For spesielt interesserte kan det jo bemerkes at i de fire siste er effekten altså ikke HELT lik. Dersom variablen f.eks. er indeksert så vil man i 'A[i+f()]+= a;' beregne indeksen bare en gang, mens man i 'A[i+f()]:= A[i+f()]+a;' beregner den to ganger.

## 4.2 Konvertering mellom aritmetiske typer

Om man skal gjøre en aritmetisk operasjon og operandene ikke har samme type konverteres først verdien med den svakeste typen (se over) automatisk til den sterkeste, og deretter gjøres operasjonen i den stekeste typen.

Man kan også angi eksplisitt konvertering. Dette gjøres ved å angi ønsket type i parentes *foran* det som skal konverteres. Ved konvertering fra reelle tall til heltall får man da trunkering ned til nærmeste heltall. For negative tall blir det trunkering "mot null", slik at '(int) -5.7' gir verdien -5.

Nødvendig konvertering ved tilordning er diskutert lenger ned.

Aritmetske konstanter skrives rett fram, og man kan også bruke "E-notasjon" for reelle tall. Merk at konstanter som er heltall og som ikke er større enn int-verdier antas å være av den svakeste typen som er mulig ut fra tallverdien, mens om man vil angi en long-konstant må den avsluttes med 'L'. Reelle konstanter antas å være av typen 'double', om man ikke avslutter dem med en 'F' for 'float'.

## 4.3 Typen char

Variable av denne typen lagrer altså tegn av ymse slag, og de lagres som Unicode i 2 byter. Konstanter av typen char angis som selve tegnet med en enkelt-apostrof på begge sider (altså som i Simula). Tall-verdien av en char-verdi kan man få ved bare å konvertere den f.eks. til int på vanlig måte, og motsatt ved å konvertere et heltall til char. Konvertering mellom char og int går ellers i stor grad automatisk, begge veier.

## 4.4 Typen boolean

Boolske uttrykk virker i Java svært likt de i Simula, og typen heter altså det samme. De viktigste perasjonene er som følger:

```
          Simula                          Java
-------------------------------------------------
         <,  <=,  >,  >=                 Som i Simula
         =,  <>                          ==,  !=
         not                             !
         and                             &
         or                              |
         and then                        &&
         or else                         ||
  excl.or (ikke egen operator i Simula)  ^
```

## 4.5 Pekere til objekter m.m.

Det som i Simula skrives 'ref(C) x, y;' (der C er en klasse) skrives i Java rett og slett 'C x, y;'. Dersom D er en subklasse av C, og E er en subklasse av D, så er følgende en lovlig if-setning:

```
if ( x instanceof D ) ... ;  // 'instanceof' er et nøkkelord i Java
```

Denne testen vil gi tilslag om x peker til et D- eller et E-objekt, men ikke om den peker til et C-objekt eller er 'null'. Den tester altså om x peker til et objekt av en subklasse (i generell forstand) av D, og den tilsvarer derved en 'in'-test i Simula (men det ser ikke ut til å være noe i Java som tilsvarer Simulas 'is'-test).

I eksempelet over er "typen til x" altså lik klassen C. Denne typen vil ha betydning f.eks. når vi skal gjøre "dot-aksess" ut fra x, eller om uttrykket inngår i en tilordning (se under). Det kan da av og til være av interesse lokalt å insistere på at 'x' er av en annen type enn C (oftest da en subklasse av C, men en superklasse er også lovlig).

Dette kan gjøres ved eksplisitt konvertering, mye på samme måte som for aritmetiske uttrykk, altså med den ønskede type (her klasse) i parentes foran variablen eller uttrykket. Man vil da få lagt inn en test på om det objektet x faktisk peker til er av en subklasse av det man konverterer til, og denne testen vil også akseptere at x er null. Om vi i klassen E har deklarert en variabel e kan vi altså passelig skrive:

```
if ( x instanceof E ) System.out.println( ((E)x).e );
```

Uttrykket '(E)x' i Java tilsvarer altså på mange måter uttrykket 'x qua E' i Simula, men med den forskjellen at det siste vil gi run-time-feil om x er none, mens det første vil tåle at x er null.

## 4.6 Tilordning

I Java brukes alltid '=' som tilordningsoperator, både for verdi-variable og peker-variable (der det altså er selve pekeren som kopieres inn i variablen på venstre side). En tilordning kan også forekomme i en parentes inne i et uttrykk, og verdien av denne parentesen er da verdien som blir tilordnet.

I visse tilfelle kreves eksplisitt konvertering:

(1) Ved tilordning mellom forskjellige aritmetiske typer får vi automatisk konvertering dersom vensteresidens type er sterkere enn høyresidens (uttrykkets) type. I motsatt fall må man angi eksplisitt konvertering av uttrykket til venstresidens type.

(2) Ved tilordning av pekere kreves eksplisitt konvertering dersom venstresiden er typet med en ekte subklasse av høyresiden.

### 4.7 Initialisering og konstanter

I Simula blir alle variable automatisk initialisert til en passelig "null-verdi". I Java gjelder dette *bare for variable som er lokale i klasser*, og verdien de får er "den opplagte" (bortsatt kanskje fra at char-variable blir initialisert med det tegn som har ASCII-verdi 0, og dette er også som i Simula).

Når det gjelder variable som er lokale i metoder eller indre blokker så gis det en annen type garanti: Kompilatoren lover å si fra dersom den ikke kan være sikker på at en variabel har fått en verdi før den brukes. I denne sammenheng er det viktig å huske på at man i alle variabel-deklarasjoner, umiddelbart etter hvert nytt variabel-navn, kan sette en tilordning som gir variablen en verdi.

Om man ønsker at en variabel aldri skal kunne forandres etter at den har fått sin verdi i selve deklarasjonen, kan man sette nøkkelordet 'final' foran deklarasjonen (og da *må* det være en tilordning i deklarasjonen).

### 4.8 "Envelope"-klasser

Av og til kan det være behagelig å betrakte verdier av de basale typene som objekter av passelige klasser. For de fleste basale typene er det derfor ferdig-definert tilsvarende klasser (med stor bokstav i navnet), nemlig klassene: Boolean, Character, Double, Float, Integer og Long. Som et eksempel ser vi på en skisse av klassen Integer:

```
class Integer{
   int val;
// Konstruktør:
   Integer(int value){val= value}
// Statiske metoder:
   static Integer valueOf  (String s) {...}
   static String  toString (int i)    {...}
   static int     parseInt (String s) {...}
// Instans-metode:
   int    intValue() {...}
}
```

## 5 Arrayer og tekster

Arrayer behandles litt annerledes i Java enn i Simula, i og med at man skiller mellom selve "array-objektet" som inneholder alle element-variablene og "array-variable" som kan peke til slike objekter. I utgangspunktet er også alle arrayer i Java én-dimensjonale, og den nedre

grensen på indeks-området er alltid null. Flerdimensjonale arrayer kan dannes ved å ha arrayer av arrayer, og det er også laget litt spesiell syntaks rundt dette.

Om vi vil ha en array der elementene f.eks. er av typen 'float' må vi altså først deklarere en variabel som kan peke til slike array-objekter, og typen av en slik variabel angis som 'float[ ]'. Generelt gjelder altså at en variabel av typen 'T[ ]' kan peke til et array-objekt der elementene er av type T. Her kan T kan være en vilkårlig type (en basal-type, en klasse, eller også en array-type).

Selve array-objektene dannes, som for andre objekter, ved new-setninger, og en slik kan generelt se slik ut: 'new T[lengde]'. Lengden kan her være et vilkårlig heltallsuttrykk, og grensene for arrayen blir da fra null til lengde-1. Man kan selvfølgelig også danne et array-objekt som initialisering til array-variablen (se eksempler under). I ethvert array-objekt finnes en heltallsvariabel 'length' som angir antall elementer objektet har plass til (og denne variablen kan bare leses, men ikke settes).

En tom array-variabel har verdien 'null'. Indeksering forgår som i Simula, bare at man bruker hake-parenteser. Noen eksempler:

```
...
float[] a= new float[100], b;
int[] c= { 2, 4, 8 }; // Tilsvarer 'new int[3]', samt verdisetting
...
b= new float[20];

for (int i= 0; i<b.length; i++) ..b[i]..; // Vil gå gjennom hele arrayen b
...
a[10]= b[19] + c[2];
...
```

Den spesielle oppramsings-varianten med krøll-parenteser for å danne et array-objekt kan bare brukes i initialiseringer.

For arrayer av typen 'C[ ]' der C er en klasse har vi litt ekstra frihet med hva slags array-objekt denne kan peke til (men dette fører dessverre også til litt ekstra run-time-testing). Om D er en subklasse av C så sier man i Java nemlig at to array-objekter som er skapt hhv. ved 'new D[...];' og 'new C[...];' står i et subklasse-forhold til hverandre. Hva dette fører til skulle fremgå av føgende eksempel:

```
class C{...}
class D extends C {...}
C[] ac;

// Følgende tre er helt greie:
ac= new C[60];
ac[10]= new C();
ac[12]= new D();

// De to første her går også bra
ac= new D[60];
ac[10]= new D();
ac[12]= new C();  // Men denne må stoppes av run-time-test
```

Vi ser altså at selv i utførelsen av den siste linja må det legges inn en run-time-test, selv om setningen tilsynelatende ser helt grei ut.

Det finnes også en alternativ syntaks for arrayer, og denne brukes en del fordi den likner mer på den som brukes i en del andre språk. Denne går rett og slett ut på at man kan si 'float a[ ], b[ ];' i stedet for 'float[ ] a, b;', og med denne konstruksjonen vil 'float a[ ], f;' gi en array-peker 'a' og en enkel float-variabel 'f'.

I utgangspunktet har ikke Java fler-dimensjonale arrayer. Man kan imidlertid lett lage dette som arrayer av arrayer, og det er også laget litt spesiell syntaks rundt dette. Følgende eksempel skulle demonstrere dette:

```
int[][] A;  //  A kan peke til en array, hvor elementene er array-pekere
A= new int[56][45]; // Spesialsyntaks: Lager mange array-objekter i ett
int i= a[7][8];  // Aksess av enkelt-variabel
```

## 5.1 Tekster

I Java finnes tre nært beslektede begreper som kan brukes til å arbeide med tekster, nemlig: (1) arrayer av typen char, (2) 'StringBuffer'-begrepet og (3) 'String'-begrepet. De to siste likner svært på typen 'char[ ]', bare at syntaksen er litt mer tekst-rettet, og at innholdet av et String-objekt ikke kan forandres etter at objektet er skapt med en viss verdi.

Eksemplene under skulle demonstrere en del omkring dette, og om hvilke konverteringer mellom disse typene som går greit. Disse eksemplene kan se litt kompliserte ut, men de er også laget for strekke grensene. Det vanlige er å bruke String-begrepet på en nokså enkel måte, og da går ting stort sett svært greit.

```
char[] cha= {'a','b','c'}, chb;
chb= new char[25];
cha[1]= chb[20];
if (i < chb.length) ...;  // 'length' er altså her en (final) variabel

StringBuffer sba= new StringBuffer("abc"), sbb;
// sbb= new StringBuffer(cha);   Virker IKKE!  Men følgende  virker,
// der '+' er konkatenering som gjør det meste om til String-verdier:
sbb= new StringBuffer("" + cha);
char c= sba.charAt(2); // Hent tegn (Vanlig indeksering virker ikke)
sbb.setCharAt(1, c);   // Sett inn tegn
if (i < sbb.length() ) ...;  // NB: 'length()' er her en METODE

String sa= "abc", sb;  // 'sa= new String("abc")' virker også.
sb= new String(cha);   // Virker faktisk!
sb= "" + sba + cha;    // Virker, men ikke uten den tomme strengen
char d= sba.charAt(2); // Hent tegn (Vanlig indeksering virker ikke)
// sb.setCharAt(1, c); Ulovlig, siden String-objekter er uforanderlige
if (i < sb.length() ) ...;  // Også her: 'length()' er her en METODE

// Alle de følgende virker:
System.out.println(cha);
System.out.println(sba);
System.out.println(sa);
System.out.println(sba + sa + cha);  // Men virker IKKE med 'cha' først!
```

Det er også diverse andre metoder knyttet til String og StringBuffer, men det går vi ikke nærmere inn på her.

# 6 Klasser og subklasser

Rekkefølgen man pedagogisk sett bør studere klasser og metoder i er litt problematisk da disse to begrepene henger så kraftig sammen. Vi regner imidlertid med at leseren har en viss kjennskap til disse begrepene på forhånd (f.eks. fra Simula). Vi går derfor først kort gjennom litt om klasser, subklasser og lokale variable, og tar så hele koblingen mellom metoder og subklasser i neste kapittel.

Grovt sett fungerer mekanismene rundt klasser og metoder begrepsmessig meget likt det tilsvarende apparatet i Simula, og den viktigste forskjellen er vel at alle ikke-statiske metoder i Java anses å være virtuelle, mens statiske metoder og alle variable aksesseres som vanlige ikke-virtuelle i Simula. Dessuten har Java statiske variable og metoder som til en viss grad "erstatter" globale variable og prosedyrer i Simula. I det følgende ser vi mer i detalj på disse tingene.

En klasse-deklarasjon har følgende form:

```
<beskrivende nøkkelord> class <klasse-navn>
 <eventuell (direkte) superklasse> <eventuell implements-liste>
 { Dekl; Dekl; ...; Dekl; }
```

De beskrivende nøkkelord kan her være: 'public' og 'abstract'/'final', der de to siste er ulogisk å angi samtidig. Synlighetskontroll ('public') diskuteres i et senere kapittel. Om 'abstract' er brukt kan man ikke lage objekter av denne klassen (men gjerne av subklasser), og man får da lov til å ha "abstrakte metodedeklarasjoner" inne i klassen (se neste kapittel). Om 'final' er brukt får man ikke lage subklasser av denne klassen.

En klasse som sådan har ikke egne parametere, men når et objekt av klassen genereres (ved 'new') startes en av de lokale metodene opp, nemlig en med samme navn som klassen selv (en slik kalles en 'konstruktør'). Derfor må man ved 'new C(..param..)' oppgi aktuelle parametere, og dette er da parametere til konstruktøren. Mer om dette i neste kapittel.

Man angir en eventuell direkte superklasse (i Java gjerne kalt en "base class") K ved å si 'extends K'. Om man ikke oppgir noe her antas klassen å være en direkte subklasse av systemklassen 'Object' (slik at en variabel deklarert: 'Object pekerVar;' kan peke til alle objekter).

En klasse kan også ha en "implements-liste", som rett og slett er nøkkelordet 'implemets', fulgt av en liste av navn på såkalte "interfacer". Interfacer er en slags "enda mer abstrakte klasser", og vi skal kort si litt om disse i et senere kapittel.

Deklarasjonene i en klasse kan være variabel-deklarasjoner og metode-deklarasjoner. Vi utsetter metode-deklarasjoner til neste kapittel, men ser her litt på variabel-deklarasjoner. Dersom en slik deklarasjon ikke er deklarert 'static' virker alt omkring den svært likt det tilsvarende i Simula. Man kan generelt aksessere slike variable ved '<objekt-peker>.<variabel-navn>', og om man er inne i klassen (eller en subklasse) klarer det seg med bare '<variabel-navn>'.

I Java (som i Simula) er det lovlig å deklarere variable med samme navn i klasser som er sub/super-klasser av hverandre. Man vil da altså alltid ved uttrykket 'OBJ.VAR', der 'OBJ' er en peker-variabel med type K, få tak i variabelen med navn VAR deklarert i klassen K eller

i en super-klasse. Selv om OBJ faktisk peker til et objekt av en subklasse KK av K og en ny variabel VAR er definert i KK, så er det altså aldri denne man får tak i.

Dersom en variabel-deklarasjon er 'static' vil den altså resultere i én variabel som er knyttet til selve klassen, og som eksisterer gjennom hele programmets utførelse uavhengig av om det blir laget objekter av klassen eller ikke. Slike kan mest logisk aksesseres ved '<klasse-navn>.<variabel-navn>' (eller bare '<variabel-navn>' innenfra klassen), men man kan også bruke <objekt-peker>.<variabel-navn>'. I siste tilfelle er det bare typen av objektpekeren som brukes, og pekeren som sådan kan godt være 'null'.

### 6.1 Bruk av 'this'

Om man inne fra en ikke-statisk metode vil ha en peker til det objektet man i øyeblikket er i, får man dette ved å bruke nøkkelordet 'this'. Rent teknisk vil denne være typet med den klassen man tekstlig er i, selv om objektet som sådan kan være av en subklasse.

## 7 Metoder, parametere og koblingen til (sub)klasser

En metode-deklarasjon ligger alltid lokalt i en klasse, og har formen:

```
<beskrivende nøkkelord> <retur-type> <metode-navn> ( <formelle parametere> )
<eventuell liste av (ubehandlede) unntak som kan oppstå inni metoden>
 { ... Sekvens av variabel-deklarasjoner og imperativer ... }
```

I Java bruker man altså ikke noe nøkkelord 'method' eller liknende. Det er eksistensen av formelle parameterene som angir at dette er en metode-deklarasjon. *Parentesen omkring parameterlisten må derfor være med selv om det ikke er parametre* (ellers forsøker kompilatoren å tolke det hele som en variabel-deklarasjon). Tilsvarende må parentesene alltid være med i kall på metoder.

En metode kan returnere et resultat, og typen av dette angis som <retur-type>. Om man ikke vil returnere noe angis her 'void'. I første tilfellet må man returnere fra metoden med setningen 'return <uttrykk>;', der uttrykket angir resultatet. I motsatt fall kan man bruke bare 'return;' eller man kan "gå ut gjennom '}' for metoden".

De beskrivende nøkkelord i en metode kan være følgende (der de med skråstrek mellom er ulogisk å angi samtidig): 'protected'/'public'/'private', 'static' og 'abstract'/'final'. Synlighetskontroll ('protected'/'public'/'private') diskuteres i et eget kapittel senere, mens 'static', 'abstract' og 'final' diskuteres straks nedenfor.

En eventuell liste av unntak skal starte med nøkkeltordet 'throws', som følges av en liste av unntaks-klasser (subklasser av klassen Throwable, se et senere kapittel) .

### 7.1 Parameteroverføring

Parameteroverføringen er i Java svært enkel: I metode-deklarasjonen ramser man opp de formelle parameterene med komma mellom, og med typen foran hver parameter. Hver formell parameter blir en lokal variabel i metoden, og som aktuell parameter for hver formell må man ha et uttrykk av tilsvarende type. Selve parameter-overføringen foregår så rett og slett som en

tilordning: '<formell>= <aktuell>', med de tilsvarende regler for automatisk konvertering. Pekere både til vanlige objekter og til array-objekter kan altså greit overføres som parametere.

Dette tilsvarer altså det som kalles value-overføring i Simula, og noen annen måte å overføre parametere på finnes ikke i Java.

Om man ønsker at man inne fra metoden ikke skal kunne forandre verdien av en formell parameter kan dette angis ved å sette 'final' foran denne parameteren.

## 7.2 Overlasting ("Overloading")

I likhet med i en del andre språk (men ikke i Simula) er det slik at man for metoder ikke bare bruker navnet, men også antallet og typen på parameterene for å identifisere hvilken metode som menes når man gjør et kall. Det betyr at det ikke regnes som ulovlig dobbelt-deklarasjon om man innen samme klasse deklarerer to (eller flere) metoder med samme navn, bare de kan skilles ad ved antallet eller typen av parameterene. De regnes da som to helt forskjellige metoder, og ved kall avgjøres hvilken som menes ut fra antallet og typen av de aktuelle parameterene. (Det er her noe snadder i forhold til automatisk konverteing i parameteroverføringen, men dette skal vi ikke gå nærmere inn på).

## 7.3 Aksess av metoder

Kall av metoder gjøres i Java med samme syntaks (altså "dot-notasjon") som man bruker ved aksess av variable i klasser, bare at listen av aktuelle parametere kommer i tillegg (og i Java *må* altså parentesene være med, selv om det er null parametere). I det generelle tilfellet er imidlertid virkningen en annen enn for variable, i og med at ikke-statiske metoder i Java aksesseres omtrent som virtuelle prosedyrer aksesseres i Simula. For statiske metoder og konstruktører gjelder spesielle regler som vi skal se på lenger ned.

Et kall på en ikke-statisk metode har altså generelt formen '<objekt-peker>.<metode-navn>(parametere)', og dette utføres i Java som følger: Først beregnes objekt-pekeren, og om pekeren er 'null' gis det run-time-feil. Ellers går men til den faktiske klassen av objektet som pekes ut og leter oppover gjennom superklassene etter en metode som stemmer med både navnet og parameterenes antall og type. Man velger så den første man finner, og gjør et vanlig kall på denne.

At dette søket opp gjennom superklassene alltid vil være vellykket garanteres av Java-kompilatoren, ved at den gjør følgende sjekk: Den ser på typen K av objekt-pekeren. Fra klassen K og opp gjennom superklassene leter den så etter en metode som passer både med navn og parametere til kallet. Om ingen slik finnes godkjennes ikke programmet. I motsatt fall vet vi at objektet som utpekes under utførelsen må være av en subklasse av K, og søket må derfor senest stoppe på den metoden som kompilatoren fant.

Innenifra en klasse kan vi selvfølgelig korte ned aksessen til bare metodenavnet og parameterene, og aksessen gjøres da som 'this.<metode-navn>(parametere)'. Merk her at selv om 'this' er typet med den klassen K man tekstlig er inne i, så vil man ikke nødvendigvis få tak i den versjonen som er synlig fra kallstedet, men kanskje én som er definert i en ekte subklasse av K.

## 7.4 Abstrakte metoder

Om nøkkelordet 'abstract' er brukt på en metode skal den bare ha parameter-angivelse, men ikke noe kropp (og den må da avsluttes med et semikolon). Den omsluttende klassen må da også være abstrakt, og dette vil gjøre at man ikke kan skape objekter av klassen. Meningen med en abstrakt metode-deklarasjon er altså å si fra at alle (ikke-abstrakte) subklasser vil ha en konkret utgave av denne metoden, men at hva de gjør vil variere, og at man i inneværende klasse ikke har noe fornuftig innhold å gi den. I Simula tilsvarer dette en virtuell-deklarasjon av en prosedyre.

Statiske metoder og konstruktører (se under) kan ikke være abstrakte.

## 7.5 Statiske metoder

Om nøkkelordet 'static' er brukt i en metode-deklarasjon betyr det, akkurat som for variable, at denne metode-deklarasjonen er knyttet til klassen som sådan, og ikke til de enkelte objektene av klassen. Det vil si at man kan gjerne kan kalle metoden uten at et eneste objekt av klassen er generert. Kall på disse metodene blir derfor behandlet noe annerledes, nemlig mer som variable, uten noen slags "virtuell effekt".

Et kall på en statisk metode har normalt syntaksen '<klasse-navn>.<metode-navn>(parametere)', og kompilatoren leter da fra den angitte klassen og opp gjennom superklassene etter en metode som passer mht navn og antall/typer på parameterene. Om den metoden man finner ikke er statisk er programmet feil, ellers bestemmer kompilatoren rett og slett at det er denne som skal kalles (uten noe slags virtuell-apparat). Helt tilsvarende blir det om man er inne i klassen og bare bruker '<metode-navn>(parametere)'.

Man kan også for statiske metoder bruke syntaksen '<objekt-peker>.<metode-navn>(parametere)', men i dette tilfellet ser kompilatoren bare på typen K av objekt-pekeren, og tolker kallet nøyaktig som om det stod 'K.<metode-navn>(parametere)'.

Siden statiske metoder ikke er knyttet til noe objekt, kan man inne fra disse bare referere (direkte) til statiske variable og statiske metoder tilhørende den klassen de er deklarert i (eller superklasser).

## 7.6 Bruk av 'super'

Dersom vi i en klasse C har en ikke-statisk metode M og i en subklasse D deklarerer en ny versjon av denne metoden, så vil det, ut fra det som er sagt til nå være umulig å få tak i C-versjonen av M i et D-object. Det er imidlertid av og til nyttig å kunne gjøre dette, og det er derfor laget en spesial-mekanisme for å få dette til. Om man fra en ikke-statisk metode i D sier 'super.M(...)', så vil man eksplisitt få tak i C-versjonen av M (på helt ikke-virtuell måte). Noen tilsvarende mekanisme finnes dessverre ikke i Simula.

Merk at 'this' ikke har en tilsvarende effekt. Om vi i klassen C sier 'this.M(...)' så vil vi i et D-objekt få tak i D-versjonen av M.

## 7.7 Bruk av 'final'

Nøkkelordet 'final' betyr nokså forskjellige ting avhengig av i hvilken sammenheng det brukes. Vi har sett at for variabel-deklarasjoner betyr det at variablen er å betrakte som en konstant.

For klasser betyr det at man ikke får lov å definere subklasser.

For metoder i klasser (både for statiske og ikke-statiske) betyr det å angi 'final' at man ikke får lov å definere metoder i subklasser som har samme navn og samme antall og typer på parameterene. Dette kan altså brukes til å garantere at brukere av en klasse vil få "klassens egen" versjon av en metode, og ikke en versjon som brukeren selv har definert i en subklasse.

### 7.8 Konstruktører

Konstruktører er spesielle metoder som har samme navn som klassen, og som kalles implisitt når man lager et objekt av klassen. En konstruktør skal ikke ha noen type-angivelse, og 'void' antas implisitt. De kan gis synlighets-angivelse, og man bør sette 'public' på konstruktørene dersom klassen er 'public'.

Det kan godt være flere konstruktører i en klasse, men de må på vanlig måte kunne skilles ad ut fra antallet/typen av parameterene. Objekter genereres ved syntaksen: 'new <klasse-navn> (parametere)', der parameter-parentesene *må* være med. Det som da skjer er at objektet dannes, og deretter kalles en konstruktør valgt ut fra antall og typen av de oppgitte parameterene.

Det er ikke mulig å kalle en konstruktør ved sitt navn som for vanlige metoder. Den eneste måten man kan få aktivisert en konstruktør (bortsett fra implisitt ved objekt-generering) er fra en konstruktør i en subklasse. Dette må da gjøres ved den spesielle syntaksen 'super(parametere);', og det må alltid gjøres *som første imperativ* i subklassens konstruktør. Dersom det ikke gjøres et slikt "super-konstruktør-kall" i starten av en konstruktør setter kompilatoren her automatisk inn imperativet 'super();'.

Det hører så med til historien at om man ikke definerer *noen* konstruktør i en klasse, så antar kompilatoren at det er definert en konstruktør uten parametere, og med kroppen '{super();}'. Konstruktøren (med null parametere) i klassen Object gjør ingen ting.

Til slutt en advarsel: Dersom man deklarerer en metode med samme navn som den omsluttende klassen, men med vanlig syntaks (altså med type eller 'void'), så blir denne akseptert av kompilatoren, men den blir IKKE betraktet som en konstruktør. Dette kan for nybegynnere virke nokså forvirrende (kan forfatteren underskrive!).

## 8 Unntaksbehandling

For å få et program som er greiest mulig å forstå er det som regel behagelig å la programlogikken følge det "normale" tilfellet, og samtidig forsøke å la behandlingen av alle slags rare tilfeller bli angitt "off line". For å kunne gjøre dette på greiest mulig måte er det i Java laget et system for å kunne fange opp og behandle spesielle situasjoner, og hovedelementene i dette er som følger:

For det første finnes et ferdig-definert lite klasse-hierarki, som er skissert under. Objekter av disse klassene er ment å bære informasjon fra det stedet der det oppstår en spesiell situasjon (i en throw-setning) til det stedet der dette blir håndtert (i et catch-alternativ i en try-setning). Man kan også fritt definere egne subklasser av disse klassene, og de vil da kunne brukes på samme måte, men kanskje bære med seg litt ekstra informasjon.

```
class Throwable{
   String message;
   void Throwable(String msg) {message= msg;}
   void Throwable() {message= "";}
   String getMessage() {return message;}
   // pluss litt mer ...
}

class Error extends Throwable{
   ... Tilsvarende konstruktører og lite mer ...
}

class Exception extends Throwable{
   ... Tilsvarende konstruktører og lite mer ...
}

class RuntimeException extends Exception{
   ... Tilsvarende konstruktører og lite mer ...
}
```

I tillegge til disse vil de forskjellige bibloteks-klasser ofte definere egne subklasser av disse, og disse må man kjenne til når man bruker ting fra biblioteket.

Objekter av klassene Error og RuntimeException (og subklasser av disse) antas å representere "system-feil", og de blir derfor noe spesialbehandlet av Java-systemet. Alle vanlige run-time-feil (f.eks. at man indekserer utenfor grensene av en array) blir angitt ved objekter av subklasser av RuntimeException. Feil som mer har med den eksekverende omgivelsen å gjøre (f.eks. at all lagerplass er brukt opp) angis ved objekter av subklasser av Error.

Spesialbehandlingen man får ved disse exception-typene er, grovt sett, at man i metode-deklarasjoner ikke behøver å ha dem med i "throws-lista" (se under). Man kan altså tenke seg at enhver metode implisitt har med Error og RuntimeException (og dermed alle deres subklasser) i sin trows-liste. Man kan godt definere egne subklasser av Error og RuntimeException, og disse vil da bli spesialbehandlet på samme måte.

En liste over alle exceptions som kan oppstå, og hvordan de er ordnet i super- og sub-klasser finnes i dokumentsjonen til biblioteks-klassene.

## 8.1 Throw-setningen og try-setningen

For å angi at en spesiell situasjon er oppdaget, og at man vil ha den behandlet, utfører man en throw-setning, som typisk kan se slik ut:

```
if (alder>130){
  throw new Exception("Verdien av 'alder' er for stor, nemlig: " + alder);
}
```

Når det skjer noe ulovlig under utførelsen (f.eks. at man indekserer utenfor en array-grense) vil det implisitt bli utført en throw-setning med et Thowable-objekt av en passelig subklasse, og med en passelig melding.

For å fange opp og behandle det som settes i gang av en throw-setning bruker man en try-setning, som typisk ser slik ut (der klassene 'Minfeil' og 'Feil' også må være deklarert som subklasser av Trowable):

```
try{
  ...
  throw new Exception(...);
  ...
  throw new MinFeil(...);
  ...
  M(...); // Antar at det i throws-lista til M står 'Feil'
  ...
}
catch (MinFeil mf)
 { ... Gjør noe med saka, kanskje bare: System.out.println(mf.getMessage);  }
catch ( Feil f)
 { ... Gjør noe med den saka ... }
catch (Exception e)
 { ... Gjør noe med den saka ... }
finally {... Eventuell felles avslutning på alle catch-alternativene ...}
```

Her utføres altså selve blokken bak 'try' på vanlig måte, men om det inne i denne blir utført en throw-setning vil systemet se på catch-alternativene i (den nærmest omsluttende) try-setninga. Disse blir forsøkt sekvensielt, og en klasse-angivelse gir her også tilslag for alle subklasser. Om man får tilslag utføres blokken bak den tilsvarende catch, og "deklarasjonen" i parentesen etter 'catch' antas da å være initialisert med det aktuelle Throwable-objektet. Legg merke til at alternativet med Exception er satt sist, ellers ville det kanskje fange opp de to andre (som man vel nettopp ville spesialbehandle).

Når en exception er ferdig behandlet utføres en eventuell 'finally'-blokk, og deretter fortsetter man etter try-setningen. Det er altså aldri mer enn én catch-blokk som utføres. Om det inne i en try-setning ikke oppstår noen exception så hopper kontrollen forbi alle catch- og finally-delene, og fortsetter på første setningen etter hele try-setningen.

Om det er oppstått en exception og denne ikke blir fanget av den nærmeste omsluttende try-setning går systemet videre til neste omsluttende try-setning osv., og om den i det hele tatt ikke blir fanget inne i den omsluttende metoden vil systemet *gå til kallstedet* for denne metoden og fortsette letingen der etter et catch-alternativ som kan behandle den.

Om systemet ikke finner noe brukerdefinert catch-alternativ som vil behandle en exception, så vil den alltid til slutt bli fanget av systemet. Når et Java-program 'xyz' utføres kan vi altså tenke oss at det er "kalt" fra operativsystemet omtrent slik:

```
try{
  xyz.main(...options etc... );
}
catch ( Throwable t ) { System.out.println(t.getMessage); }
```

Til slutt: I enhver metode-deklarasjon hører det altså med en liste av exceptions som kan bli "kastet", men som ikke blir fanget og behandlet, inne i denne metoden. I denne listen vil det å angi en klasse også dekke alle subklasser av denne klassen. Dessuten slipper man altså å ramse opp alle system-feilene (subklasser av RuntimeError og Error) hele tiden. Merk imidlertid at input/output-feil ikke regnes som system-feil i denne forbindelse.

# 9 Pakker og synlighetsregulering av navn

Ved enkle Java-programmer skriver man gjerne hele sitt Java-program på én fil, og importerer eventuelt de biblioteks-klassene man trenger, f.eks. til input/output. Man kan imidlertid også spre sitt program på flere filer, og en viktig hensikt med dette vil da som regel være at man kan kontrollere synlighet av navn mellom de forskjellige delene. Hver slik bit heter i Java en "pakke" (package), og man kan definere to typer slike pakker:

**Én-fils-pakke:** Dette er rett og slett en fil med et antall klasser, akkurat som vi har sett på over, og den vil gjerne starte med et antall import-setninger (se under). I en slik pakke kan bare én klasse være 'public', og filnavnet må stemme med navnet på denne klassen. Dersom en slik pakke skal være hovedprogrammet må klassen som er 'public' også inneholde en statisk metode 'main', slik vi har sett på tidligere.

**Fler-fils-pakke:** I en én-fils-pakke kan bare én klasse være synlig utad, og det kan av og til være lite hensiktsmessig. Java holder imidleretid hårdnakket på at i hver fil skal bare én klasse være 'public', og for å få til pakker med flere synlige klasser må de derfor tillate pakker som består av flere filer. Disse filene må da ligge under ett "directory", la oss si med navn 'mittdir' (og merk: dette navnet skal ikke ha noen 'extension'). Hver av program-filene under 'mittdir' må så ha extension '.java', og deres første setning må være 'package mittdir;' (som altså betyr at de er del av denne pakken). Hver av disse filene kan så ha én 'public' klasse, og navnet på denne må igjen stemme med navnet på filen. I en fler-fils-pakke kan man nokså fritt (uten at de er deklarart 'public') referere til navn deklarert på andre filer i samme pakka.

## 9.1 Import-setninger

Generelt kan man referere til klasser i andre pakker ved å angi fil-navnet der denne klassen er definert. Man skal imidlertid bruke punktum i stedet for '\' eller '/' i filnavnet, og men skal ikke angi '.java' på filen med selve klassen. Om det i pakka 'mittdir' er to filer 'A.java' og 'B.java', så kan man fra en annen pakke under samme directory som 'mittdir', direkte snakke om klassene: 'mittdir.A' og 'mittdir.B' (uten å bruke import-setninger i det hele tatt).

Import-setninger er laget for å kunne bruke korte klasse-navn også på klasser som ligger i andre pakker. Spesielt for biblioteks-klasser ville navnene ellers kunne bli svært lange. Import-setninger er av to typer:

```
import <navn til en java-fil (uten '.java'-extension)>;
import <navn til pakke-directory>.*;
```

I alle fil-navnene brukes her altså punktum i stedet for '\' eller '/'. Det finnes også et apparat (med bl.a. en op.sys.-variabel CLASSPATH) som angir hvor og hvordan man skal lete for å finne disse filene. Vi går ikke nærmere inn på dette her.

I den første versjonen av import-setningen angir man altså direkte navnet til en '.java'-fil, og denne kan være en én-fils-pakke eller del av en fler-fils-pakke. En slik setning medfører at man uten videre i programmet kan refere direkte til public-klassen på denne filen.

I den andre versjonen gjør vi alle public-klasser i en fler-fils-pakke direkte synlige på en gang.

Når man har flere import-setninger kan det lett skje at man får inn flere klasser med samme navn fra forskjellige pakker. Dette vil i seg selv ikke føre til protester fra kompilatoren, men

om man forsøker å *bruke* et slikt klassenavn vil den protestere (og da kan man eventuelt gå tilbake til fullstendig navning på stedet, uten import-setning).

### 9.2 Regulering av synligheten

For å regulere synligheten ( i tillegg til det som følger av skop-regler etc.) av deklarerte navn kan man foran deklarasjonen angi ett av nøkkelordene 'private', 'protected', 'public' eller man kan la være å angi noe. Regnet fra strekere til svakere restriksjoner på synligheten virker disse som følger:

Angir man **private** for en variabel eller metode i en klasse så er denne deklarasjonen bare synlig inne i denne klassen (og ikke en gang i subklasser av klassen).

Angir man ingenting om en deklarasjon så er denne synlig i hele pakken, men ikke utenfor. Man sier da gjerne at denne deklarasjonen har synlighet "**friendly**", men dette finnes altså ikke som nøkkelord.

Angir man **protected** på en variabel eller metode i en klasse K så er denne deklarasjonen synlig i hele pakken, og i tillegg i subklasser av K også om disse er deklarert i en annen pakke.

Angir man **public** om en deklarasjon så er den synlig over alt, både innenfor og utenfor den pakken den står i.

## 10 Om kompilering og kjøring av Java-programmer

Java-kompilatoren settes igang med kommandoen 'javac <progfil>.java'. For én-fils-pakker er dette greit, men for fler-fils-pakker kan man ikke be om å få kompilert hele pakka som sådan, men må kompilere de enkelte '.java'-filene. Vanligvis vil imidlertid kompilering av én av dem også føre til kompilering av de andre, så det er sjelden noe problem (og det vil vanligvis også bli gjort som en bieffekt av kompilering av hovedprogrammet, se under).

Resultatet av en kompilering blir lagt ut i et antall '.class'-filer, og disse blir lagt under det samme directoriet som den tilsvarende '.java'-filen. Det kan imidlertid bli mange '.class'-filer da hver eneste klasse i en '.java'-fil blir lagt på en egen '.class'-fil, også de som ikke er public.

Når kompilatoren i et program finner bruk av en klasse som ligger på en annen fil må den ha tak i '.class'-filen for denne klassen for å kunne fortsette kompileringen. Den leter da imidlertid både etter en .java'-fil og en '.class'-fil med det klassenavnet den er ute etter, og avhengig av hva den finner gjør den forskjellige ting:

**Den finner bare en '.class'-fil:** Da bruker den rett og slett denne.

**Den finner bare en '.java'-fil:** Da kompilerer den den til en '.class'-fil, og bruker denne.

**Den finner både en '.class'-fil og en '.java'-fil:** Da sjekker den først om '.java'-filen er nyere enn '.class'-filen, og kompilerer den i så fall til en ny '.class'-fil. Deretter bruker den den mest aktuelle '.class'-filen.

**Den finner ingen av delene:** Da gir systemet en feilmelding.

På denne måten ser man at én kompilering kan trekke med seg en mengde andre kompileringer. Ved å kompilere hovedprogrammet får man (re)kompilert alle klasser som er nødvendig for dette programmet.

En utførelse av et Java-program startes med kommandoen 'java <programnavn>'. Program-

navnet må angi en '.class'-fil (uten '.class'-extension), og den aktuelle klassen må ha en metode 'main', som så settes i gang.

Etter hvert som man under utførelsen av programmet får bruk for andre klasser hentes disse fra sine filer, og de må da alle være ferdig kompilert på '.class'-filer.

For interesserte kan nevnes at programmet ligger på '.class'-filene i såkalt "byte-kode", som er et format som er klargjort til interpretering, men som også innholder all informasjon om navn etc. Det man starter opp med kommandoen 'java <programnavn>' er en såkalt JVM (Java Virtual Machine), som er et program som kan interpretere byte-kode og hente inn '.class'-filer etter hvert som de trengs.

Siden byte-koden interpreteres går ikke denne type utførelse av Java-programmer like raskt som det man opplever med mange andre språk. Det finnes imidlertid andre måter å kompilere og kjøre Java-programmer på enn den som er omtalt her, men vi går ikke nærmere inn på det.

# 11 Bruk av interfacer

Bortsett fra at nøkkelordet 'class' er byttet ut med 'interface', så ser en interface-deklarasjon ut som en abstrakt klasse-deklarasjon som: (1) ikke har variabel-deklarasjoner, (2) ikke har konstruktører, (3) ikke har statiske metoder og (4) der alle metode-deklarasjoner er 'public' og abstrakte. På grunn av (4) får man i en interface-deklarasjon lov til å sløyfe nøkkelordene 'public' og 'abstract' om man vil (men de blir i alle tilfelle antatt å være der). Det er heller ikke lov å skape objekter av interfacer.

En interface-deklarasjon er ment å representere en "brille" som man kan se et objekt (av en "ekte klasse") gjennom. Med denne brillen kan man bare se de deklarasjonene i objektet som er listet opp i interfacen. For at et objekt skal kunne sees på med en slik brille må man i klassen til objektet ha angitt denne interfacen i "implements-listen" (se kapittelet om klasser). For at dette skal være lovlig må klassen ha definert (enten i seg selv eller i en superklasse) ikke-statiske metoder som stemmer med de i interfacen (både mht. navn og antall/typer på parameterene), og alle disse må være angitt som 'public'. Klassen kan gjerne ha flere deklarasjoner ut over dette.

Måten å bruke interfacer på er la dem være typer i variabel-deklarasjoner, f.eks. slik: 'interfA a, b;'. Variablene a og b kan da inneholde pekere til alle objekter av klasser som implementerer interfA, men med vanlig dot-notasjon kan man da bare se det som er deklarart i interfacen.

Om en klasse K implementerer en interface I, så betraktes på de fleste måter K som en subklasse av I. Om vi har deklarasjone 'I a, b;' og 'K x, y;' kan vi derfor både si: 'if (a instanceof K) ...;' og gjøre konvertering (casting) slik: 'x= (K)a;'

Interfacer kan ikke selv implemetere andre interfacer, men de kan derimot ha en extend-liste som ikke bare inneholder én, men flere andre interfacer (som den da altså arver innholdet til). Detaljene omkring hvordan dette fungerer skal vi imidlertid ikke gå inn på her.