Java Programmering

En bog for begyndere

Skrevet af Henrik Kressner

Indholdsfortegnelse

Introduktion

Denne "Javabog" er et hastigt sammenkog af den jeg skrev i 1998. Meget er fastholdt, og meget vil blive ændret, der er sket en del siden 1998. Selv om grundprincipperne er identiske, så er specielt brugerinterfacet nærmest overalt blevet grafisk.

Det er tanken bogen skal skrives helt om, så den i stor grad vil bruge Netbeans som værktøj.

Bogen kan frit benyttes af alle der kan finde anvendelse for den, det er selvfølgelig venligt at kreditere hvor man tager tingende fra.

Kommentarer? Send en mail til forfatteren: kressner@synkro.dk

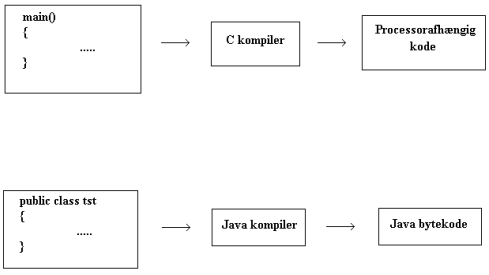
Morud Marts 2014

Henrik Kressner

1 Introduktion til Java

Java blev udviklet i starten af halvfemserne som et værktøj til at gøre programmering af vaskemaskiner, ure, autocomputere og lignende mere ensartet og enkel. Det var ikke nogen umiddelbar succes, hvilket imidlertid ændrede sig, da Internettet blev udbredt. Det viste sig nemlig, at såkaldte Java applet's kunne bruges til at gøre hjemmesider mere smarte, med bevægelig grafik og lignende.

Java er et objektorienteret programmeringssprog, der har lånt en del fra det udbredte objektorienterede programmeringssprog C++, der igen bygger på det meget udbredte ANSI C. Java skal fortolkes af en Javafortolker, en såkaldt Javaengine. Dette gør, at Java er platform uafhængig. Platformuafhængigheden opnås ved at Java ikke kompileres til en processorbestemt binærkode, men til en såkaldt bytekode. Bytekoden skal så fortolkes af en Javaengine, placeret på den maskine, Javaprogrammet skal eksekveres på.



Figur 1.1

Fig 1.1 Viser forskellen mellem en normal kompiler, og den teknik der benyttes i Java. Den kode der kommer ud af en Javakompiler, er altid den samme, hvorimod den kode der kommer ud af en C kompiler, er afhængig af den processor, den er kompilet til.

1.1 Javakoden

I forbindelse med Webbet, har Java et par store fordele. Java er kompilet og fylder meget mindre end HTML kode, hvilket betyder kortere overførselstider. Derudover er Java et avanceret programmeringssprog der kører på brugerens (clientens) computer, og derfor ikke bruger processorkraft på serveren. Denne funktionalitet er nok Java's største fordel, og kan åbne helt nye perspektiver, da Java i realiteten er et distribueret programmeringssprog.

Java er blevet forenklet i forhold til C++, idet de faciliteter, der ofte giver anledning til fejl i C og C++ programmer, er fjernet. Til gengæld er der tilføjet andre faciliteter. Følgende faciliteter er taget ud af Java i forhold til C++:

- Pointer.
- Operator overstyring.
- Strukturer.
- Union’s.
- Multibel nedarvning.

Java er blevet udbygget med følgende faciliteter:

- Flertrådet.
- Monitorer til synkronisering.

Disse faciliteter medfører til gengæld mulighed for nye fejl. Først og fremmest race problemer i forbindelse med synkronisering mellem tråde. Disse problemer kan igen medføre, at et program hænger, men det kommer vi tilbage til.

1.2 Det første program

I dette kapitel starter vi helt fra bunden. Alting bliver gjort så enkelt som muligt, herved gives der mulighed for alle kan følge med. For hvert eksempel er der en kildetekst (sourcecode), og en beskrivelse af det output der opstår når programmet eksekveres. Dette kan virke trivielt for personer der har megen programmeringserfaring. Begynderen kan til gengæld støtte sig op ad det.

1.2 Skriv til skærmen.

Vi vil starte med et lille program, der kun har en opgave her i verden: at skrive en linie på skærmen. Kildeteksten (også kaldet koden eller sourcecode) er givet i Fig 1.2.1a, output fra programmet er givet i Fig 1.2.1b

```
// Filnavn = hallo1.java
// Bemærk, filnavnet skal have java som extendet.
// Dette program udskriver "Jeg er, ergo er jeg"
// på skærmen. Programmet kan ikke køre i en
// browser, det kræver tekstmode

public class hallo1
{
        public static void main(String args[])
        {
                System.out.print("Jeg er, ergo er jeg");
        }
}
```

```
C:\java hallo1
Jeg er, ergo er jeg
C:\
```

Fig 1.2.1a

Programmet hallo1 vil kunne køre, hvis man indtaster kildeteksten (Fig 1.2.1a) i en ASCII editor, (Tekstbehandling er udemærket, men husk at gemme som en ASCII fil) og gemmer den med filnavnet *hallo1.java*. **Intet andet filnavn kan accepteres**.

Derefter skal programmet kompiles. Hvis JDK'et benyttes, kaldes javac kompileren med filnavnet hallo1.java som argument, dette skal se således ud.

javac hallo1.java

Herved skaber kompileren en fil ved navn *hallo1.class*. Denne fil indeholder Java bytekoden. Bytekoden kan normalt ikke eksekveres direkte og derfor skal bytekoden startes med Javaengine på denne måde:

java hallo1

Bemærk at filextendet *class* skal udelades.

Programmet starter med fem linier der begynder med to skråstreger `//` . To skråstreger betyder det efterfølgende på denne linie er en kommentar, og kompileren skal ignorere det. Kommentarer bruges til at tydeliggøre kildeteksten, således at vi mennesker kan forstå hvad der skal ske.

Det er en god ide at bruge kommentarer i kildeteksten. Årsagen er: Når man evt. skal redigere en kildetekst, man har skrevet for et halvt år siden, kan det være svært at huske hvad man tænkte da man skrev programmet. Strategisk velplacerede kommentarer kan gøre det nemmere at læse en kildetekst, både for en selv og andre.

Derefter følger linien: `public class hallo1`. Dette er en klasseerklæring. Hvad det mere præcist betyder, kommer vi tilbage til. På nuværende tidspunkt skal vi blot vide, at den skal være der. Det er vigtigt at klassenavnet, (her hallo1) er præcis det samme som filnavnet, uden java extendet.

Bemærk: Java er case sencetive, hvilket betyder, at for Java er der forskel på store og små bogstaver. I dette eksempel ville det altså være en fejl, at skrive:

```
public class Hallo1
```

Klassens indhold skal være samlet inde i `{}` (kaldes start og slut tuborg parenteser på grund af udformningen). Inde i selve klassen har vi en såkaldt metode. I dette tilfælde er det main( ) metoden der er erklæret som `public static void`, og som tager argumentet args af typen String[]. Hvad alt dette betyder, vil vi også komme tilbage til. Lige nu er det bare noget vi gør, for at komme i gang.

Metoden main( ) har sit indhold mellem de følgende tuborg's. I dette eksempel kalder vi print metoden, der udskriver en linie på skærmen.

I kildeteksten kan de to ovenstående afsnit forklares som vist på Fig 1.2.2, blot ved at bruge kommentarer:

```
// Filnavn = hallo1.java
// Bemærk, filnavnet skal have java som extendet.
// Dette program udskriver "Jeg tænker, ergo er jeg"
// på skærmen. Programmet kan ikke køre i en
// browser, det kræver tekstmode

public class hallo1   // KLasseerklæring
{       // Klassens starttuborg
        public static void main(String args[])    // main metoden
        {       // main metodens starttuborg
                System.out.print("Jeg tænker, ergo er jeg");
        }       // main metodens sluttuborg
}       // Klassens sluttuborg
```

```
C:\java hallo1
Jeg tænker, ergo er jeg
C:\
```

Fig 1.2.2

På Fig 1.2.2 kan man se, at kommentarer også kan stå på samme linie som andet. Det er blot vigtigt, at kommentaren står sidst på linien, da man ellers udkommenterer dele af kildeteksten. Figuren er også et eksempel på en kildetekst med lige lovligt mange kommentarer. Men hellere for mange kommentarer end for få, efterhånden som du bliver dygtigere, finder din brug af kommentarer et naturligt leje.

For nuværende er kernen i vores program linien:

```
System.out.print("Jeg tænker, ergo er jeg");
```

Vi vil nu prøve at se hvad man kan med en beslægtet metode. (Bemærk: metode er et udtryk der bruges i Javaprogrammering. En metode kan sammenlignes med en funktion i C/C++ og procedurer/funktioner i Pascal/Delphi)

```
// Filnavn = hallo2.java
// Dette program udskriver "Jeg tænker, ergo er
// jeg" på skærmen efterfulgt af et linieskift

public class hallo2
{
    public static void main(String args[])
    {
        System.out.println("Jeg tænker, ergo er jeg");
    }
}
```

```
C:\java hallo2
Jeg tænker, ergo er jeg

C:\
```

Fig 1.2.3

I koden på Fig 1.2.3 har vi benyttet metoden `System.out.println("Jeg tænker, ergo er jeg")`, i stedet for `System.out.print("Jeg tænker, ergo er jeg").` Den beskedne forskel (ln i print) medfører, at vi nu får et linieskift efter linien er udskrevet.

(Rent faktisk hedder metoderne print og println. Begrebet System.out. der står foran, kommer vi tilbage til senere)

Koden i Fig 1.2.3 kan kompiles med kommandoen *javac hallo2.java* og kan eksekveres med kommandoen *java hallo2*.

En anden måde at bruge print og println metoderne, er vist i Fig 1.2.4.

```
// Filnavn = hallo3.java
// Dette program deler udskriften over flere linier
// i kildeteksten

public class hallo3
{
    public static void main(String args[])
    {
        System.out.println("Jeg tænker, "+
        "ergo er jeg");
    }
}
```

```
C:\java hallo3
Jeg tænker, ergo er jeg

C:\
```

Fig 1.2.4

Dette er for at vise, at man kan skrive mere end et argument med print metoden, blot ved at benytte + (plus) tegnet. Man kan endda skrive videre på næste linie, man skal blot huske at adskille med et plus. Dette betyder dog ikke at man kan benytte plustegnet til alting. (Plustegnet er faktisk en operator. Mere om det senere)

En forkert måde at bruge plus operatoren på er vist i Fig 1.2.5.

```
// Filnavn = hallo4.java
// FEJL !!!
// Programmet kan ikke kompiles

public class hallo4
{
    public static void main(String args[])
    {
        System.out.println("Jeg tænker, ergo er jeg +
        Jeg er stadig");
    }
}
```

Fejl!!!

Programmet i dette eksempel kan ikke kompileres, da tekststrengen ikke er afsluttet før linieskiftet. (Der mangler et “ på hver linie)

Fig 1.2.5

Hvis man forsøger at kompilere koden fra Fig 1.2.5, vil man få en fejlmeddelelse i stil med:

Hallo4.java:9:String not terminated at end of line.

Dette betyder, at kompileren har fundet en fejl i linie 9, og fejlen er en streng ikke er afsluttet med gåseøjne, før et linieskift.

Opgave 1.2.1 Skriv et program der udskriver dit for- og efternavn på én linie

Opgave 1.2.2 Skriv et program der udskriver dit fornavn på en linie, og dit efternavn på den efterfølgende linie.

1.3 Variabler

Variabler er et sted, hvor vi kan opbevare og manipulere data. Vi vil senere komme nærmere ind på begrebet, lige nu vil vi blot vise et simpelt eksempel på brug af 3 variabler.

```
// Filnavn = var1.java
// Demonstrerer variabler

public class var1
{
public static void main(String args[])
    {
         int a,b,c;    // Her erklæres 3 variabler

         a = 1;    // Her tilskrives variablen a med værdien et.
         b = 2;    // Her tilskrives variablen a med værdien to.
         c = 3;    // Her tilskrives variablen a med værdien tre.
         System.out.print("a = " + a + " b = " + b + " c = " + c);
    }
}
```

```
C:\java var1
a = 1 b = 2 c = 3
C:\
```

Fig 1.3.1

Koden i Fig 1.3.1 har 3 variabler ved navn a, b og c, de er af typen `int`. (integer = heltal) Med linien:

```
int a,b,c;
```

erklærer vi variablerne. Det betyder at vi føder dem, og reserverer plads i computerens hukommelse. Efter en tom linie tilskriver vi de 3 variabler med værdierne 1, 2 og 3. I den sidste linie, (vores printmetode) udskriver vi indholdet af variablerne på skærmen, kombineret med en forklarende tekst. Det ser således ud:

```
System.out.print("a = " + a +" b = " + b + " c = " +  c);
```

Den forklarende tekst er omgivet af `“”`. (gåseøjne) Det betyder, at det der står inde i gåseøjnene skal udskrives på skærmen, som det står. Dette er en såkaldt tekststreng. Det der ikke er omgivet med gåseøjne, er variabler. Det medfører, at printmetoden ikke udprinter et a, men udprinter det som der står i a. I dette tilfælde tallet 1.

Variabler findes som forskellige typer. Dette kommer vi også tilbage til.

Det er kutyme, at man laver et linieskift efter variabelerklæringer. Det gør man kun for at skabe overblik for os mennesker, kompileren er ligeglad.

Vi kan bruge variabler til at opsamle resultatet af operationer. Dette er vist på Fig 1.3.2, hvor variablen c opsamler summen af variablerne a og b.

```
// Filnavn = var2.java
// Viser tilskrivning af variabler

public class var2
{
        public static void main(String args[])
        {
                int a,b,c;    // Variabel erklæring
                a = 1; b = 2;    // Initiering af to variabler
                c = a + b;       // Tilskrivning af variablen c

                System.out.print("c = " + c);
                System.out.print(" a = " + a + " b = " + b);
    }
}
```

```
C:\java var2
c = 3 a = 1 b = 2
C:\
```

Fig 1.3.2

Programmet var2.java på Fig 1.3.2 minder en del om var1.java, forskellen ligger i linierne:

```
a = 1; b = 2;       // Initiering af to variabler
c = a + b;          // Tilskrivning af variablen c
```

I den første linie tilskrives variablen a med værdien 1 og variablen b med værdien 2. Denne form for tilskrivning kaldes også for en initiering. Dette skyldes, at det er første gang de to variabler bliver tilskrevet med en konstant værdi. (Vi vil senere gøre det på en smartere måde)

I den anden linie bliver variablen c tilskrevet med summen af variablerne a og b. Det betyder, at uanset hvad der før stod i variablen c, så står der nu værdien tre. Man kunne i stedet have skrevet:

```
c = 1 + 2;        // Tilskrivning af variablen c
```

Dette viser, at plus operatoren tager to operander, nemlig tallet 1 og tallet 2. Lighedstegnet er også en operator, det tager kun en operand, nemlig resultatet af operationen 1 + 2.

Hvis vi forestiller os udtrykket:

```
c = a + b – 5;
```

kan vi se, at minustegnet tager to operander, (b og tallet 5), og afleverer resultatet af subtraktionsoperationen som den ene operand til plus operatoren. Plus operatoren lægger denne operand til indholdet af den anden operand (variablen a). Resultatet af denne operation bruges som operand for lighedstegnet, der aflever resultatet i variablen c.

Programmet var3.java på Fig 1.3.3 viser et eksempel på, at man kan tage resultatet af en operation (her en sum), og bruge resultatet som input til noget andet. I dette tilfælde printmetoden.

```
// Filnavn = var3.java
// Viser at resultatet af en beregning
// kan bruges direkte.

public class var3
{
    public static void main(String args[])
    {
         int a,b;

         a = 1; b = 2;
         System.out.print("Summen af a og b er : ");
        // Næste linie udskriver summen af a og b
         System.out.print(a + b);
         // Bemærk: Indholdet af a og b er uændret.
         System.out.print(" a = " + a + " b = " + b);
    }
}
```

```
C:\java var3
Summen af a og b er: 3 a = 1 b = 2
C:\
```

Fig 1.3.3

Vi vil senere vende tilbage til begrebet operatorer og operander.

Opgave 1.3.1 Skriv et program der kan trække to tal fra hinanden, og afleverer resultatet på skærmen.

Opgave 1.3.2 Skriv et program der indeholder variablerne omsaetning, vOmk, fOmk og resultat. Programmet skal, efter variablerne er initialiseret, udskrive et årsregnskab.

Tip: DB (Dækningsbidrag) = omsætning – variable omkostninger (vOmk). Resultatet = dækningsbidrag – faste omkostninger.

1.4 Intelligent

En computer er ikke så intelligent som mange gør den til, men et af de træk der kan kaldes intelligent, er if … else strukturen. Dette betyder kort og godt, at hvis (if) et eller andet er sandt, så udfører man en ting, ellers (else) udfører man noget andet. Det hele afhænger altså af, om programmøren kan stille nogle brugbare spørgsmål, af den type der kun kan svares ja eller nej til.

Programmet if_1.java viser et eksempel på brug af betingelser i Java.

```
// Filnavn = if_1.java

public class if_1
{
       public static void main(String args[])
       {
              int a, b, c = 10;
              a = 5; b = 10;
              c = a * b;

              if (c > 40)
                     System.out.println("c er større end 40");
              else
                     System.out.println("c er mindre end 40");
              System.out.print("c = " + (a * b));
    }
}
```

```
C:\java if_1
c er større en 40
c = 50
C:\
```

Fig 1.4.1

Der er et par nye begreber i koden Fig 1.4.1. For det første tilskriver vi en variabel (i dette tilfælde variablen `c`) med værdien `10`, samtidig med den bliver erklæret. Dette kaldes at initialisere en variabel. Vi kunne have valgt at initialisere samtlige variabler i en arbejdsgang ved at skrive

```
int a = 5, b = 10, c = 10;
```

Så kunne vi slippe for at tilskrive dem senere i koden. Vi kunne også have valgt at skrive det således.

```
int a = 5;
int b = 10;
int c = 10;
```

Derudover begynder vi at regne lidt i dette program. Linien `c = a * b;` betyder, at vi tager indeholdet af `b` (som er `10`) og ganger det med indholdet af variablen `a` (5), resultatet placerer vi i variablen `c`, der altså nu har værdien 50. Indholdet af variablerne `a` og `b` ændres ikke, men `c` er ikke længere 10, men 50.

Selve if sætningen fungerer på følgende måde. Hvis betingelsen er sand, `(c > 40),` så udføres linien `System.out.println("c er større end 40");` ellers (else) udføres linien `System.out.println("c er mindre end 40");`

Linien `System.out.print("c = " + (a * b) );` udføres under alle omstændigheder, da den står uden for if sætningen. Man kan I øvrigt argumentere at det er en meget dårlig programmeringsteknik der er benyttet her. Årsagen er, at produktet af variablerne `a` og `b` ikke længere behøves være lig med indholdet af `c`, en eller begge variabler kunne jo være blevet ændret af en anden programdel.

En teknik der ikke kunne give anledning til fejl ville være:

```
System.out.print("c = " + c );
```

Dette udtryk vil altid vise c's værdi på skærmen, og det er jo det udtrykket påstår det gør.

I Programmet if_2.java er dette blevet rettet, samtidigt er programmet gjort lidt simplere, selv om det i bund og grund er det samme program. Man kan jo argumentere, (og det er ikke altid lige smart), at hvis programmet ikke fortæller mig at en betingelse er sand, så må betingelsen jo være falsk.

```
// Filnavn = if_2.java

public class if_2
{
        public static void main(String args[])
        {
                int a, b, c = 10;
                a = 5; b = 10;
                c = a * b;

                if (c > 40)
                        System.out.println("c er større end 40");
    }
}
```

```
C:\java if_2
C er større end 40

C:\
```

Fig 1.4.2

Programmet udskriver kun at `c` er større end fyrre, hvis `c` er er større end fyrre. Hvis `c` ikke er større end fyrre, så får brugeren intet at vide. I nogle tilfælde kan det være udemærket, i andre tilfælde kan det give anledning til misforståelser.

Der er mange andre måder at misforståelse kan opstå, derfor er det vigtigt at programmøren gør det let for sig selv at læse kildeteksten. En af metoderne til at forbedre læsbarheden, er at undgå magiske tal. I de hidtidige eksempler vi har vi brugt linien

```
if (c > 40)
```

Her er tallet fyrre et magisk tal. Det kaldes sådan fordi ingen kan huske hvorfor det står der. Måske indgår tallet fyrre endda flere steder i programmet og i andre sammenhænge.

I programmet if_3.java har vi afskaffet det magiske tal, ved at indføre en variabel ved navn `graense`, og sætte den lig med fyrre. Herefter kan vi ændre if sætningen til det mere (for mennesker) meningsfyldte:

```
if (c > graense)
```

Blot ved at se på det, kan man se, det er en mere læseværdig måde at skrive en betingelse på. Det har også en anden fordel. Forestil dig, at du har brugt tallet fyrre flere steder i dit program. En dag ændres grænseværdien til 50. Hvad gør du? Går du ind og leder efter alle de steder hvor der står fyrre, eller retter du bare linien `graense = 40;` til `graense = 50;`?

```
// Filnavn = if_3.java
// Fjernelse af magiske tal

public class if_3
{
       public static void main(String args[])
       {
              int graense = 40;
              int a, b, c = 10;

              a = 5; b = 10;
              c = a * b;

              if (c > graense)
                     System.out.println("c er større end " + graense);
              System.out.print("c = " + c);
   }
}
```

```
C:\java if_3
C er større end 40
C:\
```

Fig 1.4.3

Der findes en del måder at udtrykke en betingelse i Java. På nuværende tidspunkt kan vi klare os med at kende følgende betingelser:

a == b	Er a lig med b ?
a != b	Er a forskellig fra b ?
a > b	Er a større end b ?
a < b	Er a mindre end b?
a <= b	Er a mindre end eller lig med b ?
a >= b	Er a større end eller lig med b ?

Resultatet af en betingelse skal altid være logisk sandt (true), eller logisk falsk (false).

Opgave 1.4.1 Skriv et program der udskriver indholdet af den største af to variabler.

1.5 Rundt og rundt

En grundliggende programmeringsteknik, er gentagelse. Med gentagelse menes, at en ting (en programdel) udføres igen og igen, indtil et tilfredsstillende resultat er opnået. Dette resultat skal kunne udtrykkes som en betingelse, der kan være enten sand eller falsk.

```
// Filnavn = while_1.java
// Udskriver alle hele tal
// i intervallet 1 ... 9

public class while_1
{
        public static void main(String args[])
        {
                int graense = 10;
                int i = 1;

                while ( i < graense)
                {
                        System.out.println(" i = " + i);
                        i =  i + 1;
                }
        }
}
```

```
C:\java while_tst
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
C:\
```

Fig 1.5.1

Programmet while_1.java kan tælle fra 1 til 9. Programmet starter med at definere en grænseværdi. I dette tilfælde er den 10. Derefter initialiseres variablen `i` til værdien 1.

Så går programmet ind i while linien, og tester om variablen `i` er større end variablen graense, som er 10. Da variablen i = 1, hvilket er mindre end 10, går vi ind i løkken. Her udskrives værdien af variablen i til skærmen. Derefter lægges der en til variablen `i`, og størrelsen af variablen `i` testes igen i while's betingelse. Når `i` bliver større end 9, falder betingelsen for at gå ind i løkken, og programmet termineres.

Linien `i = 10` vil aldrig blive udskrevet. Dette skyldes, at når variablen `i` bliver lig med 10, så falder betingelsen, og while stopper. Da println metoden er skrevet inde i whileløkken, (mellem start- og slut-tuborg), bliver den kun udført hvis while løkkens betingelse er sand.

Hvis programmet skal køre frem til ti, kan man flytte linien `i = i + 1;` op foran kaldet til println metoden. Det vil dog få den effekt, at i = 1 ikke vil blive udskrevet.

Bemærk at whileløkken udfører det, der står på linien efter løkken. Hvis man ønsker at udføre mere end en linie, som i programmet while_1.java, så skal man ramme det man ønsker at udføre inde i whileløkken, ind i en tuborgparenteser.

Der bør ikke være noget semikolon efter et while udsagn. Hvis man alligevel sætter et, risikerer man at programmet går ind i en evig løkke.

```
while (betingelse); // FEJL !!! Kompilerer fint, men vil "hænge"
```

Programmet while_2.java viser, hvordan man med en simpel gentagelse kan slippe for et trivielt stykke arbejde.

```
// Filnavn = while_2.java
// Dette program udskriver den lille tabel

public class while_2
{
    public static void main(String args[])
    {
        int max = 10;
        int i = 1;

        System.out.println(" Den lille tabel:");
        while ( i <= max)
        {
            System.out.print(" 10 * " + i);
            System.out.println(" : " + (i * 10));
            i =  i + 1;
        }
    }
}
```

```
C:\java while_2
Den lille tabel:
10 * 1 : 10
10 * 2 : 20
10 * 3 : 30
10 * 4 : 40
10 * 5 : 50
10 * 6 : 60
10 * 7 : 70
10 * 8 : 80
10 * 9 : 90
10 * 10 : 100
C:\
```

Fig 1.5.2

Man kunne vælge at udskifte linien

```
i = i + 1;
```

med linien

```
i = i + 10;
```

så ville det samme program lave en lidt anderledes tabel.

Hvis man glemmer linien:

```
i = i + 1;
```

Så vil programmet hænge. Det er en ting de fleste programmører nok oplever i ny og næ, så det vil vi fortsætte med at arbejde med, på næste side.

Problem opstår ikke så ofte, når man bruger en anden løkkestruktur - `for` løkken - som vi nu vil se på.

```
// Filnavn = for_1.java

public class for_1
{
    public static void main(String args[])
    {
        int graense = 10;
        int i;

        for (i = 1; i < graense; i = i + 1)
            System.out.println("i = " + i);
    }
}
```

```
C:\java for_1 i
= 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9

C:\
```

Fig 1.5.3

Med for løkken opfordres programmøren, til at huske tilskrivningen af kontrolvariablen. Derfor er det ikke så nemt at glemme tilskrivningen, men der er stadig mulighed for at gå ind i evige løkker.

Programmet for_1.java viser et enkelt eksempel på brug af en for løkke. Det centrale i dette program er linien:

```
for (i = 1; i < graense; i = i + 1)
```

Dette betyder, at når programeksekveringen møder linien, starter den med at sætte variablen i = 1. Hvis variablen `i` ikke er erklæret inden, opstår en kompilerings fejl. Derefter kontrolleres, om `i` er mindre end `graense`. Hvis dette er tilfældet, og kun da, vil løkken blive eksekveret. Til sidst i løkken vil udsagnet `i = i + 1` blive udført.

Bemærk I øvrigt, at der ikke bruges tuborg parenteser i forbindelse med dette eksempel. Det skyldes ene og alene, at der kun skal udføres en ting inde i for løkken; nemlig udskrivning til skærmen. Variablen `i` vil blive opdateret, hver gang løkken gennemløbes, præcis som i while eksemplet. Forskellen er blot, at i en for struktur er opdatering af kontrolvariablen automatiseret, i while skal programmøren huske den.

Faktisk kan man godt ændre koden fra programmet for_1.java til

```
for (i = 1; i < graense; )
{
      System.out.println("i = " + i);
      i = i + 1;
}
```

Det vil virke på præcis samme måde som før.

Det er dårlig programmeringsteknik, at røre ved en kontrolvariabel inde i en for løkke. Årsagen er,

at al kontrol høre hjemme inde i `for` udtrykket. Begynder programmøren at pille ved sine kontrolvariabler, inde i for løkker, mister programmøren nemt overblikket. Typisk med et periodisk hængende program som resultat.

Programmet for_2.java viser et eksempel på en kombination af en for løkke, og en if sætning.

Linien: `int midt = max  / 2;` er ny, men betyder bare, at variablen midt er af typen `int` (heltal), og den initieres med værdien max divideret med 2, altså max halve. Hvis max havde haft en ulige værdi, eksempelvis 11, ville divisionen stadig virke. Dette skyldes at vi for nuværende kun arbejder med hele tal. Hvis der opstår en rest i en division, bliver den ganske enkelt smidt væk.

```
// Filnavn = for_2.java

public class for_2
{
    public static void main(String args[])
    {
        int max = 10;
        int midt = max  / 2;
        int i;

        for (i = 1; i < max; i = i + 1)
        {
            System.out.println("i = " + i);
            if (i == midt)
                System.out.println("Vi er midtvejs");
        }
    }
}
```

```
C:\java for_2
i = 1
i = 2
i = 3
i = 4
i = 5
Vi er midtvejs
i = 6
i = 7
i = 8
i = 9

C:\
```

Fig 1.5.4

Den sidste løkkestruktur der findes i Java, er do … while løkken. Den største forskel mellem do … while løkken, og de to forrige løkkestrukturer er, at en do … while løkke altid gennemløbes mindst en gang. Dette skyldes at betingelsen testes i slutningen af løkken.

Programmet do_while1.java viser, hvordan man kan tælle til 9 med en do … while løkke. Et sjovt eksperiment er at se, hvad der sker hvis man bytter om på linierne

```
System.out.println("Der var " + i);
i = i + 1;
```

Således at der i stedet kommer til at stå

```
i = i + 1;
System.out.println("Der var " + i);
```

Ved at lave denne lille ændring, vil man opnå at tælle fra 2 til 10.

```
// Filnavn = do_while1.java

public class do_while1
{
    public static void main(String args[])
    {
        int i;

        i = 1;
        do
        {
            System.out.println("Der var " + i);
            i = i + 1;
        }
        while (i < 10);
        System.out.println("i er nu = " + i);
    }
}
```

```
C:\java do_while1
Der var 1
Der var 2
Der var 3
Der var 4
Der var 5
Der var 6
Der var 7
Der var 8
Der var 9
i er nu = 10

C:\
```

Fig 1.5.5

Opgave 1.5.1 Omskriv programmet for_2.java, således at det i stedet bruger en while løkke.

Opgave 1.5.2 Omskriv programmet for_2.java, således at det i stedet bruger en do … while løkke.

Opgave 1.5.3 Tilpas programmet do_while1.java, således at programmet tæller fra et, til og med 10.

1.6 Slut på starten

Dette har været en kort gennemgang af nogle få Javateknikker, for at få begynderen i gang, og den mere øvede til at se hvordan man arbejder i Java. Efter denne korte tur rundt i Java, vil jeg anbefale læseren, den øvede såvel som den uerfarne, at kigge på de opgaver der afslutter kapitlet, for at sikre sig at stoffet fra dette kapitel sidder fast.

Inden vi helt forlader det grundlæggende, er her et par programmer der har et par sjove effekter. Den uøvede kan springe disse eksempler over, og evt. komme tilbage til dem senere. Først er der programmet rente1.java.

```
// Filnavn = rente1.java
// Tæller til 10

public class rente1
{
    public static void main(String args[])
    {
        int terminer,i;
        double rente, startsum, slutsum;

        terminer = 10;
        rente = 3.5;
        slutsum = startsum = 1000;
        for (i = 1; i <= terminer; i = i + 1)
            slutsum = slutsum * (1 + rente/100);
        System.out.print("Efter " + terminer);
        System.out.println(" terminer er summen " + slutsum);
    }
}
```

```
C:\java rente1
Efter 10 terminer er summen
1410.598760621121

C:\
```

Fig 1.6.1

Her indføres en ny datatype kaldet `double`. Variabler erklæret som `double` (i dette tilfælde `rente, startsum og slutsum`), kan indeholde kommatal, modsat variabler af typen `int`, der kun kan indeholde hele tal. Årsagen til at vi vælger at bruge double er, at renter jo netop udregnes som brøkdele af et grundtal, og sådanne brøkdele kan ikke opbevares i et heltal.

I næste kapitel vi gøre mere ud af int, double og andre datatyper.

Programmet rente1.java foretager en simpel renteberegning, der kunne udtrykkes matematisk som:

$$\text{Slutsum} = \text{startsum}(1 + \text{rentefod})^{i}$$

Vi kan altså udføre et trivielt stykke matematik ved en simpel gentagelse.

Programmet sekperdoegn.java viser hvordan man, på en ganske besværlig måde, kan beregne antallet af sekunder i et døgn. For at gøre beregningen tydelig, er der valgt at bruge fire variabler. Man kan argumentere for, at variablerne `sek`, `min` og `timer` er overflødige, men de er med til at gøre det nemmere for programøren at læse kildeteksten.

Den kvikke læser vil straks kunne se, at i stedet for at skrive:

```
for (h = 0; h < timer; h = h + 1)
     for (j = 0; j < min; j = j + 1)
          for (i = 0; i < sek; i = i +1)
               sekunder = sekunder + 1;
```

Kunne man have skrevet:

```
Sekunder = sek * min * timer;
```

Ved at flette løkkerne ind i hinanden har vi altså opnået en multiplikation.

```
// Filnavn = sekperdoegn.java
// Programmet beregner antallet
// af sekunder i et døgn

public class sekperdoegn
{
    public static void main(String args[])
    {
        int sek = 60;
        int min = 60;
        int timer = 24;
        int sekunder = 0;
        int h, i, j;

        for (h = 0; h < timer; h = h + 1)
            for (j = 0; j < min; j = j + 1)
                for (i = 0; i < sek; i = i +1)
                    sekunder = sekunder + 1;
        System.out.print("Antallet af sekunder i et døgn er : ");
        System.out.println(sekunder);
    }
}
```

```
C:\java sekperdoegn
Antallet af sekunder i et døgn
er : 86400
C:\
```

Fig 1.6.2

Opgave 1.6.1 Tilpas rente1.java således at det udføres med en while løkke.

Opgave 1.6.2 Tilpas rente1.java således at det udføres med en do … while løkke.

Opgave 1.6.5 Tilpas programmet sekperdoegn.java således at der bruges en for løkke, en while løkke, og en do … while løkke.

1.7 Opgaver

1.7.1 Skriv et program der tæller baglæns fra 10 til nul.

1.7.2 Tilpas programmet i 1.7.1 således, at når værdien er under 3, skal der udskrives "Stopper snart" på skærmen.

1.7.3 Tilpas programmet i 1.7.2 således, at når værdien er 3, skal der udskrives "Så er der ikke langt igen" på skærmen.

1.7.4 Skriv et program der udskriver et vandret histogram, bestående af asterisker (*). En for hver værdi af et tal, der kan være i intervallet 0 ... 70.

1.7.5 Skriv et program der udskriver et lodret histogram, bestående af asterixer (*). En for hver værdi af et tal, der kan være i intervallet 0 ... 20.

1.7.6 Tilpas 1.7.4 således at der udskrives histogram for to tal i det givne interval.

1.7.7 Tilpas 1.7.5 således at der udskrives histogram for to tal i det givne interval.

1.7.8 Skriv et program der beregner hvor langt du kommer efter 1, 2 .. 9 timers kørsel med hastigheden 100 km/t. Resultatet skal udskrives på skærmen.

1.7.9 Skriv et program der udskriver alle lige tal mellem 1 og 10.

2 Grundbegreber

Efter at have fået et førstehåndsindtryk af Java i kapitel 1, vil vi nu gennemgå nogle klassiske programmeringstekniske grundbegreber.

2.1 Blokke

Blokke bruges til at fortælle kompileren hvilke dele af programmet der hører sammen. En blok starter altid med en { (kaldes start tuborg) og slutter altid med }. (slut tuborg)

Foreløbig har vi benyttet blokke uden at tænke ret meget over dem, hvilket har begrænset os en smule. Studer koden blok1.java i Fig 2.1.1:

```
// Filnavn = blok1.java

public class blok1
{
    public static void main(String args[])
    {
        int i;
                for(i = 0; i < 10; i = i + 1)
                   System.out.println("Udskrives 10 gange");
                System.out.println("Udskrives 1 gang");
    }
}
```

```
C:\java blok1
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 1 gang

C:\
```

Fig 2.1.1

Det fremgår klart af teksten hvad der forventes at ske, og det er da også oplagt. Men hvad nu hvis der i stedet havde stået som vist i programmet blok2.java?

```
// Filnavn = blok1.java
public class blok2
{
    public static void main(String args[])
    {
        int i;

        for(i = 0; i < 10; i = i + 1)
            System.out.println("Udskrives 10 gange");
            System.out.println("Udskrives også 10 gange");
    }
}
```

```
C:java blok2
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives 10 gange
Udskrives også 10 gange

C:\
```

Fig 2.1.2

Her vil det oplagte ikke ske. Forvirringen skyldes indrykningen i linien:

```
System.out.println("Udskrives også 10 gange");
```

Der i denne sammenhæng er misvisende. Dette skyldes at en `for` løkke, betragter den efterfølgende linie som det der skal udføres inde i løkken. Hvis næste linie er et start blok mærke (en start tuborg), så vil hele den afmærkede blok blive udført for hver omgang i løkken.

Hvis vi ville have det oplagte i blok2.java til at ske, skal der indsættes blokmærker for at fortælle kompileren hvad der skal udføres for hver omgang i `for` løkken. Det kan se som vist på Fig 2.1.3.

Hvis man skal følge den kodestandard der er udgivet af Sun bør start tuborg placeres på samme linie som `for` løkken, efterfulgt af en tabulering. Jeg mener at det er mere tydeligt at se en blok hvis start og slut tuborg står over hinanden, da en blok lettere kan erkendes, uden at man først skal se hvad der står på en linie. Efter kodestandarden ville `for` løkken i blok3.java se således ud:

```
for(i = 0; i < 10; i = i + 1) {
     System.out.println("Udskrives 10 gange");
     System.out.println("Udskrives også 10 gange");
}
```

Der er ingen forskel på funktionaliteten i de to eksempler. Den eneste årsag til at vælge den ene eller den anden teknik er, at gøre kildeteksten mere læselig for mennesker.

```
// Filnavn = blok3.java
public class blok3
{
    public static void main(String args[])
    {
        int i;
        for(i = 0; i < 10; i = i + 1)
        {
            System.out.println("Udskrives 10 gange");
            System.out.println("Udskrives også 10 gange");
        }
    }
}
```

```
C:\java blok3
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
Udskrives 10 gange
Udskrives også 10 gange
C:\
```

Fig 2.1.3

I denne bog vil jeg bruge teknikken med blokmærker over hinanden når der er plads, og flytte start tuborg en linie op når det kniber med pladsen. Uanset metode, så vil slut tuborg altid stå lodret under den linie hvor blokken begynder, og alt inde i blokken vil være indrykket.

Det er ikke indrykningen der skaber blokken, vi kunne også have skrevet:

```
for ( i= 0;  i < 10; i = i + 1) { System.out.println(“Udskrives 10 gange”); System.out.println(“Dette udskrives også 10 gange”); }
```

Dette ville kompileren opfatte som det samme. Men vi stakkels mennesker kan kun med besvær se hvad der står.

En anden egenskab ved at benytte blokke er rækkevidde, kaldet scope på engelsk. Begrebet rækkevidde dækker over hvor langt væk en variabel kan ses, i forhold til hvor den blev erklæret.

Programmet scope1.java på Fig 2.1.4 erklærer to variabler, kaldet a og b. Variablen a rækker fra den bliver erklæret, og indtil den møder slut tuborg mærket for den blok hvor den blev erklæret. For variablen b gælder det samme, hvilket medfører, at b kun er synlig i den inderste blok.

```
// Filnavn = scope1.java
public class scope1
{
    public static void main(String args[])
    {
        int a = 10;

        System.out.println("a = " + a);
        {
            int b = 13;

            System.out.println("a = " + a + " : b = " + b);
        }
//      System.out.println("a = " + a + " : b = " + b);
//      Udkommenteret for at undgå kompilerfejl
    }
}
```

```
C:\java scope1
a = 10a = 10 : b = 13

C:\
```

Fig 2.1.4

Da b kun er synlig i den inderste blok, er det nødvendigt at benytte udkommentering af den sidste println metode, da der ellers vil opstå en kompilerfejl i retning af:

scope1.java:15: Undefined variabel b:

Hvilket betyder, at kompileren har fundet en fejl i kildeteksten linie 15, og at denne fejl skyldes at der bruges en ikke erklæret variabel. Dette er korrekt, da b er uden for rækkevidde.

Når en blok oprettes inden i en anden blok, kaldes det nestning. Det der gælder i en blok, gælder også i de blokke som blokken nester, men det modsatte er ikke tilfældet, som vi så i Fig 2.1.4

Rækkevidden er årsagen til jeg foretrækker at have de korresponderende blok mærker til at stå med samme indrykning, det gør efter min opfattelse rækkevidden tydeligere.

2.2 Datatyper

Datatyper benyttes til at fastslå hvilken type af data vi ønsker at opbevare i variabler af den givne type. Nogle programmeringssprog tager ikke det med typer så nøje, andre som Pascal og C kræver ret præcis angivelse af typer.

Java's typer har meget til fælles med C's typer, men er modsat C's typer defineret meget præcist. Java tilbyder følgende typer:

Type	**Plads i bit**	**Art**
byte	8	Heltal
short	16	Heltal
int	32	Heltal
long	64	Heltal
float	32	IEEE 754 Enkelt præcision, flydende komma
double	64	IEEE 754 Dobbelt præcision, flydende komma
char	16	Unicode characterer
boolean		Boolsk værdi true eller false. (Sand eller falsk)

Tabel 2.2.1

Læsere med erfaring fra andre programmeringssprog vil nok mangle string og array. Det er ikke en fejl at de ikke er med i skemaet, da disse ikke er variabler i Java, men klasser.

Alle typer i tabel 2.2.1, undtaget de to sidste, er numeriske typer. At en type er numerisk betyder, at alle variabler af den type indeholder tal.

Typen char kan indeholde en karakter, eller et tegn som vi nok vil sige på dansk. En karakter er et bogstav eller et af de tegn der kan findes på et tastatur, ikke blot vores danske tastaturer, men alle tastaturer. Det der kan virke forvirrende er, at '1' er ikke et tal men et tegn. (Det ses af de omgivende '') 1 derimod er tallet et.

I en computer er alle data i bund og grund blot tal, derfor er alle tegn også repræsenteret som tal inde i computeren, men ved eksempelvis at erklære en variabel til at være af typen char, kan programmøren bestemme hvordan programmet skal opfatte dataene.

Programmet typecnv1.java viser hvordan Java håndterer chartypen, selv om bogstavet P inde i computeren er repræsenteret ved en talværdi, udskrives bogstavet P på skærmen.

```
// Filnavn = typecnv1.java

public class typecnv1
{
    public static void main(String args[])
    {
        int i = 10;
        char a = 'P';
                System.out.println("a = " + a + "
: i = " + i);
     }
}
```

```
C:\java typecnv1
a = P : i = 10

C:\
```

Fig 2.2.1

Programmet typecnv2.java viser bogstavet P talværdi, ved at gøre brug af en cast. En cast benyttes til at konvertere variabler af en datatype, til variabler af en anden type. Når man gør brug af en cast skal man være opmærksom på, at data kan gå tabt. Det sker hvis man konverterer data fra eksempelvis en char til en byte, da byte typen kun indeholder det halve antal bit af hvad char indeholder.

I programmet typecnv2.java ville der blive mistet data, hvis variablen `i` indeholdt en værdi større end 65.536, da 16 bit jo netop giver dette mulige antal kombinationer. Da variablen `i` er tilskrevet værdien 80, mistes der ikke data, i dette tilfælde.

```
// Filnavn = typecnv2.java

public class typecnv2
{
    public static void main(String args[])
    {
        int i = 80;
        char a = (char) i;     // Her udføres et cast

        System.out.println("a = " + a + " : i = " + i);
     }
}
```

```
C:\java typecnv
a = P : i = 80

C:\
```

Fig 2.2.2

Variabler af en type kan altså konverteres til variabler af en anden type ved at bruge en teknik kaldet cast.

Opgave 2.2.1 Skriv et program der udskriver den numeriske værdi for alle de store bogstaver.

2.3 Operatorer og operander

Operatorer er nogle hyggelige småfyre der udfører operationer på et antal operander, og derefter afleverer et resultat. Nogle typiske operatorer er plus (+), minus (-), gange (*) og division (/). Fælles for disse er, at de er binære operatorer. De kaldes binære fordi de altid kræver to operander. Unære operatorer tager en operand og tertiære operatorer tager tre operander. I Appendiks A er der lister over alle Java's operatorer, her vil jeg blot demonstrere et par nyttige. For det første er der den unære operator ++, den bruges til at lægge en til indholdet af en variabel, dette er vist i programmet op1.java vist på Fig 2.3.1.

```
// Filnavn = op1.java
public class op1
{
    public static void main(String args[])
    {
        int i = 0;

        while( i++ < 5)
            System.out.println("i = " + i);
        System.out.println("i = " + i);
    }
}
```

```
C:\java op1
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6

C:\
```

Fig 2.2.3

Programmet starter med at erklære og initiere variablen `i` til værdien nul. While løkken tester om `i` er mindre end 5, hvilket er tilfældet første gang, derefter lægges der en til i og indholdet af løkken udføres, i dette tilfælde er det println metoden. Bemærk at den første værdi der udskrives er et ettal, og ikke et nul

Det korte af det lange er, at når betingelsen er skrevet således:

```
i++ < 5
```

Så vil det først blive undersøgt om `i` er mindre end fem, derefter bliver der lagt en til `i`, og løkken udføres. Når `i` bliver fem (eller mere), falder betingelsen, og løkken gennemføres ikke.

Vi kan lave lidt om på programmet, ved at ændre `i++` til `++i`, det er vist i programmet op2.java.

```
// Filnavn = op2.java
public class op2
{
    public static void main(String args[])
    {
        int i = 0;

        while( ++i < 5)
            System.out.println("i = " + i);
        System.out.println("i = " + i);
     }
}
```

```
C:\java op2
i = 1
i = 2
i = 3
i = 4
i = 5

C:\
```

Fig 2.3.2

Forskellen på de to teknikker er, at når betingelsen er:

```
++i < 5
```

Så vil der blive lagt en til variablen `i`, før det undersøges om `i` er mindre end fem, det betyder at løkken stopper en tand tidligere end før. Minus operatoren kan bruges på samme måde. Hvis man skriver `++i`, eller `--i` kaldes det præfiks notation, hvis man skriver `i--` eller `i++` kaldes det postfiks notation.

En anden nyttig operator er modulus operatoren (%), den afleverer den rest der opstår ved en division. Dette er specielt nyttigt til at tælle hvor mange gange et tal går op i et andet, da netop i det tilfælde er resten nul. Programmet op3.java viser denne teknik.

Modulus operatoren er specielt nyttig når der skal konverteres mellem talsystemer, eller hvis man ønsker at foretage sig noget ved et specielt interval.

I Java er der en anden teknik til at tilskrive variabler med. Det sker ofte i et program at man vil skrive noget i stil med:

```
graense = graense + step;
```

For at spare fingerspidserne mod for megen tasten, kan man i stedet skrive:

```
graense += step;
```

Disse to udtryk har samme funktionalitet, men det sidste er unægtelig hurtigere at skrive. Teknikken kan overføres til andre operatorer, således gælder:

```
a = a * 10;
```

er det samme som:

```
a *= 10;
```

```
// Filnavn = op3.java

public class op3
{
    public static void main(String args[])
    {
        int i = 0;

        while( i < 100)
        {
            if (i % 10 == 0)
                System.out.println("i = " + i);
            i++;
        }
        System.out.println("i = " + I);
     }
}
```

```
C:\java op3
i = 0
i = 10
i = 20
i = 30
i = 40
i = 50
i = 60
i = 70
i = 80
i = 90
i = 100
C:\
```

Fig 2.3.3

Opgave 2.3.1 Skriv et program der udskriver alle lige tal i intervallet 22 .. 38. Begge tal inklusive.

Opgave 2.3.2 Skriv et program der kan omregne en tid opgivet i minutter, til timer og minutter.

Opgave 2.3.3 Skriv et program der omregner en tid opgivet i minutter, til minutter og sekunder.

2.4 Kommandolinieargumenter

Hidtil har vi afprøvet noget kode, uden den store indsigt i hvad den gjorde, vi har bl.a. benyttet linien:

```
public static void main(String args[])
```

i alle eksemplerne. Hvis noget blev udeladt ville vi have fået en fejl under kompileringen. Objektet `args[]` er et array af objekter der hver indeholder en tekststreng. Lad os se hvad dette objekt kan bruges til, ved at prøve at køre følgende lille program.

```
// Filnavn = kom_arg1.java
// Program til udskrivning af
// kommandolinie argumenter

public class kom_arg1
{
     public static void main(String args[])
     {
        for (int i=0; i < args.length; i++)
            System.out.println(args[i]);
     }
}
```

```
C:\java kom_arg1 tst1 tst2
tst1
tst2

C:\
```

Fig 2.4.1

Programmet tager altså de argumenter vi tilføjer efter programnavnet på kommandolinien, og skriver dem ud én linie ad gangen, efterfulgt af et linieskift. Udtrykket:

```
i < args.length
```

betyder at metoden length i objektet `args` returnerer hvor mange elementer der er i array'et. Hvis dette er mindre end `i`, så stopper udførelsen. Klasser og metoder vender vi kraftigt tilbage til.

En anden måde at udføre det samme på, kunne være at ændre eksemplet fra en `for` løkke, til en `while` løkke, det kunne se ud som vist i Fig. 2.4.2.

Som det kan ses er output nøjagtigt det samme, vi har blot opnået det på en anden måde. Dette er et godt eksempel på at en `for` løkke altid kan udskiftes med en `while` løkke, det er bare ikke altid praktisk. I dette tilfælde har vi måttet tilføje linien:

```
int i = 0;
```

```
// Filnavn = kom_arg2.java// Program til
udskrivning af
// kommandolinie argumenter
// med en while løkke.

public class kom_arg2
{
        public static void main(String args[])
        {
                int i = 0;
                while (i < args.length)
                {

System.out.println(args[i]);
                        i++;
                }
        }
}
```

```
C:\java kom_arg1 tst1 tst2
tst1
tst2

C:\
```

Fig 2.4.2

før `while`. Derudover har vi været nød til at bruge en blok, da linien:

```
i++
```

ellers først ville blevet opdateret efter løkken var udført, men da variablen `i` ikke ville være blevet opdateret, så ville betingelsen `i < args.length` altid være sand, og vi ville være i en uendelig løkke.

Man kunne i stedet have valgt at skrive:

```
while (i < args.length)
 System.out.println(args[i++]);
```

Dette havde haft det samme resultat. Men hvis man ændrer tilskrivningen af `i` fra at være præfiks til at være postfiks vil det går galt. Det første element i `args` vil ikke blive udskrevet, til gengæld vil der blive forsøgt udskrevet et mere element mere end der er, hvilken vil medføre en `java.lang.ArrayIndexOutOfBoundsException`

Opgave 2.4.1 Omskriv programmerne i Fig 2.4.1 og Fig 2.4.2 således at de udskriver kommandolinieargumenterne i omvendt rækkefølge.

Opgave 2.4.2 Skriv et program der udskriver antallet af argumenter, hvis der er nogen, og skriver at der mangler argument, hvis der ikke er nogen argumenter.

2.5 Metoder

Metoder er det samme som det man i andre programmeringssprog kalder procedure og funktioner. Årsagen til at de kaldes metoder i Java er, at Java er et objektorienteret sprog der, modsat C++, har gjort sig helt fri af arven fra de procedure og funktions opdelte sprog.

En metode har et navn, det kan tage et antal argumenter, og det kan returnere en variabel der kan ignoreres af den kaldende metode. Lad os kigge på en metode, der returnerer true hvis den modtager et skudår, og returnerer false hvis det ikke var et skudår.

Inden vi går i gang skal vi lige huske, at skudår er delelige med 4, men ikke med 100, med mindre det altså er deleligt med 400. Efter denne regel er år 1900 ikke et skudår, år 2000 er et skudår, fordi det er deleligt med 400.

```
// Filnavn skudaar1.java
// Program til bestemmelse af skudår

public class skudaar1
{
   public static boolean sk_aar(int aar)
   {
      if ( (aar % 400) == 0)
          return true;
      else if ( (aar % 4) == 0)
          if ( (aar % 100) != 0)
               return true;
      return false;
   }

   public static void main(String a[])
   {
      int aar = 1900;

      if (sk_aar(aar))
         System.out.println(aar + " er et skudaar");
      Else
         System.out.println(aar + " er ikke et skudaar");
   }
}
```

```
C:\java skudaar1
1900 er ikke et skudaar

C:\
```

Fig 2.5.1

Det eneste nye i main er, at vi ikke direkte spørger om returværdien er sand. Tidligere ville vi nok have skrevet:

```
if (sk_aar(aar) == true)
     .....
```

Dette er ikke nødvendigt, men ganske korrekt. Årsagen til at vi kan udelade sammenligningen er, at en betingelse altid skal være boolsk sand eller falsk, og `sk_aar()` metoden er netop af typen boolean, og returnerer derfor en boolsk værdi vi kan teste direkte på.

Ud over main har vi tilføjet en ny metode, vi har kaldt den sk_aar for skudår. Metoden tager ét argument af typen int, i variablen `aar`. Metoden returnerer værdien boolean som allerede beskrevet.

I eksemplet har vi erklæret to variabler med navnet `aar`, én gang i main således:

```
int aar = 1900;
```

og én gang i sk_aar på denne måde:

```
public static boolean sk_aar(int aar)
```

Dette er to forskellige lokale variabler. Variabler siges at være lokale i metoden, når deres rækkevidde er metoden.

Det kan være uheldigt at have to variabler med samme navn, men da de har forskelligt rækkevidde, og da det er et naturligt navn i denne sammenhæng, kan det forsvares.

Selve `if` delen i sk_aar kan godt virke lidt uoverskuelig, dette kan man afklare ved at markere samhørigheden med blokke. Man kunne eksempelvis gøre således:

```
if ( (aar % 400) == 0)
{
    return true;
}
else
{
     if ( (aar % 4) == 0)
     {
          if ( (aar % 100) != 0)
          return true;
     }
}
return false;
```

Hermed bliver konstruktionen lidt tydeligere, men der står præcis det samme.

Der er dukket et nyt begreb op ... `return`. Når programmet under eksekvering møder en `return`, så vil al yderligere eksekvering i den metode stoppe, metodens returværdi vil blive sat til det der opgives i return linien, og dette kan der så testes på i den kaldende metode.

Bemærk: Der skal ikke nødvendigvis bruges lighedstegn i forbindelse med return.

Ud fra dette kan vi fastslå, at hvis metoden sk_aar modtager en værdi i variablen `aar`, der er delelig med 400, så vil metoden returnere true, ellers, hvis `aar` er delelig med 4, og ikke med 100, så returneres også true, ellers vil false blive returneret.

Man kunne selvfølgelig også finde ud af, at der jo rent faktisk står:

Hvis 400 går op i årstallet, eller hvis (4 går op i årstallet og 100 ikke gør det) så skal værdien true afleveres, eller skal værdien false afleveres.

Koden for dette er vist i Fig 2.5.2

```
// Filnavn = skudaar2.java
// Program til bestemmelse af skudår

public class skudaar2
{
   public static boolean sk_aar(int aar)
   {
       if ( (aar % 400 == 0) || (aar % 4 == 0 && aar %
100 != 0))
              return true;
       else return false;
   }

   public static void main(String a[])
   {
      int aar = 1904;

      if (sk_aar(aar))
          System.out.println(aar + " er et skudaar");
      else
          System.out.println(aar + " er ikke et skudaar");
   }
}
```

```
C:\java skudaar2
1904 er et skudaar

C:\
```

Fig 2.5.2

Dette er mere overskueligt, når vi ved at && er en operator der betyder and (og), og at `||` er en operator der betyder (or) ellers.

Koden i Fig 2.5.1 og Fig 2.5.2 udfører det samme arbejde, men på hver sin måde. Begge funktioner bruger `return` til at returnere en værdi. Dette skal en metode gøre, hvis den ikke er erklæret som `void`, men det kigger vi mere på senere.

Hvis vi erstattede main programmet i skudaar2.java med:

```
public static void main(String a[])
{
           int aar;

           for(aar = 1995; aar < 2005; aar++)
           if (sk_aar(aar))
                      System.out.println(aar + " er et skudaar");
           else
                      System.out.println(aar + " er ikke et skudaar");
}
```

Kan vi udføre en lille nem test af sk_aar metoden.

Opgave 2.5.1 Skriv et program der inderholder en metode til at teste om et tal er lige. Metoden skal returnere true hvis den modtager et lige tal, og false hvis den modtager et ikke lige tal, herunder nul.

Opgave 2.5.2 Skriv et program der indeholder metoden `hvor_langt(int hastighed, int tid)`. Metoden skal returnere hvor langt man er kommet, efter `tid` minutters kørsel med `hastighed` km/t.

Efterprøv programmet med følgende linie:

```
System.out.println(“hastighed = 100 og tid = 60 “ + hvor_langt(100,60) );
```

Opgave 2.5.3 Skriv et program der udskriver hvor langt man er kommet efter 14 minutters kørsel med 60 km/t, efterfulgt af 32 minutters kørsel med 80 km/t, og afsluttes med 1½ times kørsel med 47 km/t.

Tip: Brug metoden fra Opgave 2.5.2

Opgave 2.5.4 Skriv en metode `minutterTilSekunder(int minutter, int sekunder)`. Metoden skal omregne `minutte`*r* + `sekunde`*r* til sekunder, og returnere den værdi i en int.

2.6 Overloadning

Det er muligt at have flere metoder med det samme navn, blot deres argumentliste varierer, dette kaldes overloadning.

Fig 2.6.1 viser et eksempel på overloadning af metoden adder.

```
// Filnavn = overload.java
// Program til demonstration af overloadnings teknik

public class overload
{
   private static void adder(int a, int b)
   {
      System.out.println("Summen af heltallet er : "+ (a + b));
   }

   private static void adder(float a, float b)
   {
      System.out.println("Summen af det flydende tal er : " +
      (a + b));
   }

   public static void main(String args[])
   {
      int a = 10;
      int b = 13;
      float c = 15;
      float d = 25;

      adder(a,b);
      adder(c,d);
   }
}
```

```
C:\java overload
Summen af heltallet er : 23
Summen af det flydende tal er 40.0
C:\
```

Fig 2.6.1

I dette eksempel bruges void som returtype på de overloadede metoder, hvilket betyder at de ikke skal aflevere nogen værdi. I dette tilfælde ville det være en fejl at forsøge at aflevere en værdi med return.

Bemærk: En metode kan ikke overlades, blot ved at være forskellige på returtypen, forskellen skal fremgå af metodens argumentliste. Følgende vil give en kompilerfejl:

```
int tst()
{
     ...
}

double tst()
{
     ...
}
```

Det skyldes at kompileren ikke kan kende forskel ud fra metodens returtype.

Hvis metoden `tst` skal overloades kan det gøres således.

```
double tst(int test)
{
        ...
}

double tst(char alfa, int tal)
{
        ...
}
```

Hvilket skyldes at der er forskel på metodernes argumentliste. At metodernes returtyper er ens i dette tilfælde har ingen betydning.

2.7 Opgaver

Opgave 2.7.1 Skriv et program der har tre metoder for subtraktion. Hvis programmet får argumenter af typen int, skal det subtrahere to interne integer's. Hvis programmet modtager argumenter af typen float, skal det i stedet subtrahere to interne float's. Hvis programmet modtager argumenter af typen double, skal det subtrahere to interne double's. Metoden sub skal overlades til at håndtere dette. Resultatet skal udskrives på skærmen.

Opgave 2.7.2 Skriv et program der indeholder metoderne:

```
long sqr(int x,)
float sqr(float x)
double sqr(double x).
```

De tre metoder sqr() skal returnere x^2. Programmet skal udskrive værdien af x^2, i intervallet 1 .. 5 på 3 forskellige linier.

Opgave 2.7.3 Tilpas programmet fra opgave 2.7.2 således, at output kommer til at fremstå som vist på denne Fig 2.7.3.1

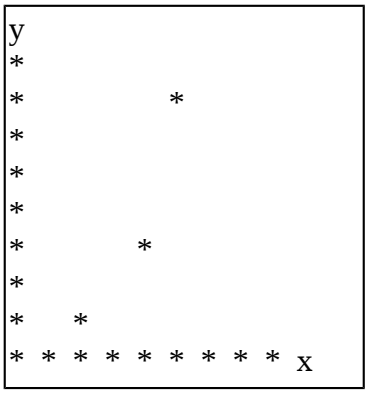


Fig 2.7.3.1

Tip: Begynd bagfra.

Opgave 2.7.4 Skriv et program der indeholder metoderne:

```
short power(byte x, byte y)
int power(short x, short y)
long power(int x, int y).
```

De tre metoder `power()` skal returnere x^y. Programmet skal udskrive værdien af x^y, i intervallet 1 .. 5 på 3 forskellige linie.

3 Klasser og objekter

Hidtil har vi bevæget os i én klasse uden at tænke nærmere over hvad der er objekter, og hvad der er klasser. Lad os starte med definitionen på en klasse og et objekt.

Definition: *En klasse er en definerbar type*

Et objekt er et tilfælde (instans) af en klasse.

En klasse er altså en definerbar type. Der findes andre typer i Java (int, double m.m.) forskellen mellem dem og en klasse er, at klasser er en type der kan indeholde både data (variabler) og programkode (metoder).

På Fig 3.1 er `minklasse` en klasse, og `mitobjekt` oprettes som et tilfælde (instans) af `minklasse`.

```
public class minklasse
{
     ...
}

....

     int a;         // a er en variabel af typen integer

     minklasse mitobjekt = new minklasse();

     // mitobjekt er et objekt af typen minklasse
     ...
```

Fig 3.1

Det er måske nærliggende, at blande en klasse sammen med en metode. Den afgørende forskel på en klasse og en metode er, at med en klasse kan man skabe et tilfælde (instans, og dermed et objekt) En metode kalder man. En metode i Java er altid en del af en klasse. Metoder tilhører altså en klasse.

En metode vil altid kræve plads i hukommelsen, en klasse kræver ingen plads, men det eller de objekter der skabes af klassen vil tage plads i hukommelsen.

Man kan sammenligne klasser og objekter med fotografiets verden, ved at betragte klassen som et negativ. (der findes ét, og kun et negativ) Med negativet kan vi skabe mange billeder. (objekter)

3.1 Skab et objekt

Hidtil har vi skabt et objekt per program vi har skrevet, uden at tænke nærmere over det. Nu vil vi lave et program der indeholder to objekter. For at sammenligne med fotografien, kalder vi den klasse vi vil skabe et tilfælde af for negativ. Objektet vil vi kalde for positiv.

```
// Et tomt program til demonstration af klasser

class negativ
{
     // Dette udvider vi senere
}


public class objekt1
{
     // Her kommer der også til at ske noget.
}
```

Fig 3.1.1

Hvis vi gemmer denne kildetekst i en fil ved navn `objekt1.java` og derefter Kompilerer den, skabes der to *.class filer. Den ene fil vil som hidtil få navnet `objekt1.class`. Derudover vil en ny blive skabt af kompileren ved navn `negativ.class.`

Efter at have kompileret kildeteksten fra Fig 3.1.1 kan vi altså konstatere, at Java altid skaber én *.class fil per klasse. Filens navn er det samme som klassenavnet, efterfulgt af class.

Vi vil nu skabe et tilfælde (objekt) af negativ, og kalde det positiv. Dette er vist i Fig 3.1.2

```
class negativ
{
}



public class objekt2
{
        negativ positiv = new negativ();
}
```

Fig 3.1.2

Ved at kompilere denne fil vil der blive skabt et objekt ved navn positiv, der er af typen negativ. Objektet bliver skabt i klassen objekt2. Dette medfører, at objekt2 kan benytte metoder fra klassen negativ.

Vi vil nu tage det sidste spring, og erklære metoder i begge klasser, og bruge metoderne fra negativ klassen i objekt klassen.

```
// Filnavn = objekt3.java
// Kode til demonstration af
// klasser og objekter

class negativ
{
        public void print(int a)
        {
                System.out.print(a);
        }
}

public class objekt3
{
        public static void main(String args[])
        {
                int lokal = 22;
                negativ positiv = new negativ();

                positiv.print(lokal);
        }
}
```

```
C:\java objekt3
22
C:\
```

Fig 3.1.3

Metoden `main()` er i klassen `objekt3`. Dette objekt oprettes via kaldet fra kommandolinien. (java objekt3) Når programmet køres skabes der først en variabel af typen `int`, denne tilskrives værdien `22`. Derefter skabes der et objekt at typen (klassen) `negativ`. Dette nye objekt får navnet `positiv`.

I sidste linie af main() benyttes metoden print i objektet positiv, hvilken medfører et kald til print metoden i objektet positiv. Hvis vi ikke havde startet med at erklære et tilfælde af negativ, kunne vi ikke bruge metoden print.

Hvis vi havde forsøgt at benytte print metoden på denne måde:

```
negativ.print(lokal);    // Fejl !!!
```

Ville kompileren give en fejlmeddelelse, fordi man ikke uden videre kan bruge en metode fra en klasse, der skal først skabes et objekt fra klassen.

Opgave 3.1.1 Skriv et program der indeholder to klasser *loekke* og *udskriv*. I *loekke* skal *main()* metoden være. I *udskriv* skal der være en metode *void lige()*, og en metode *void ulige()*. Metoden *lige()* skal udskrive, "Tallet er lige", metoden *ulige()* skal udskrive "Tallet er ulige" når de kaldes. I *main()* metoden skal en løkke gennemløbes fra 0 til 16. Hvis løkkens kontrolvariabel er lige, skal metoden *lige()* kaldes, er kontrolvariablen ulige skal *ulige()* metoden kaldes.

Opgave 3.1.2 Skriv et program der indeholder to klasser *regn* og *brug*. I klassen regn skal der være følgende fire metoder:

```
int adder (int a, int b)
int subtraher (int a, int b)
int gange (int a, int b)
int divider (int a, int b)
```

I klassen brug skal main metoden være. I main skal der oprettes et tilfælde af regn, og de fire regnearter skal afprøves.

Opgave 3.1.3 Udvid main metoden i 3.1.2 såæedes at der oprettes tre tilfælde (objekter) af regn klassen. De tre objekter skal navngives regn1, regn2 og regn3. Test funktionalitetetn med følgende linie i main:

```
System.out.print(regn1.adder(regn2.adder(1,1), regn3.adder(1,1));
```

Opgave 3.1.4 Udvid 3.1.2 med de tre metoder:

```
int areal_rektangel(int laengde, int hoejde)
int areal_trekant(int laengde, int hoejde)
int areal_cirkel(int radius)
```

3.2 Indkapsling

En stærk bevæggrund til at man i sin tid udviklede objektorienterede programmeringssprog er, at man kan øge abstraktionsniveauet. Dette opnås ved at indkapsle data og metoder inde i objektet. Dette kan sammenlignes med, at man kan bruge en telefon uden at vide noget om elektronik og telekommunikation. Kommunikationsteknikken er indkapslet i plastik, input foregår i mikrofonen, output kommer ud af en lille højtaler. Hvordan det lader sig gøre er irrelevant for brugeren, ergo befinder brugeren sig på et højere abstraktionsniveau end udvikleren.

Man kan udvidde diskussionen til, at man behøves ikke at være brobygningsingeniør for at kører over en bro, man skal blot kunne køre bil. Man behøves ikke at kunne svejse for at køre cykel, det problem har en anden løst for os når vi køber cyklen.

Inden for programmering gælder det om at indkapsle teknikken på samme måde, så brugere af et objekt (og det kan godt være os selv) ikke skal spekulere over hvordan en opgave bliver løst, men blot kan koncentrere sig om at løse opgaven.

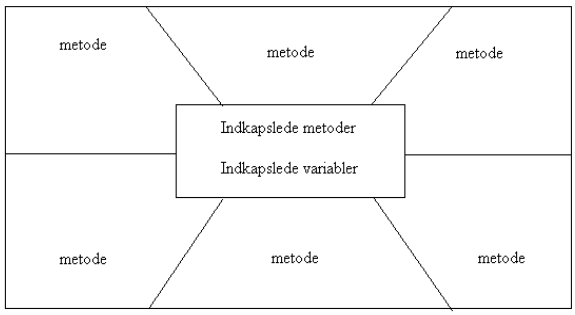


Fig 3.2.1

Fig 3.2.1 viser princippet i indkapsling. Hele figuren er ét objekt. Inderst i objektet er der nogle metoder og variabler der kun kan ses inde i objektet, dette kaldes private metoder og variable. I en skal uden om disse er interfacet, d.v.s. de metoder der kan ses uden for objektet. Udefra objektet kan man ikke manipulere med de inderste metoder og variabler, hvis man på nogen måde ønsker at nå dem, skal det ske gennem interfacets metoder.

Der er ingen grund til at overveje at lade metoder uden for objektet pille ved de inderste metoder og variabler, det er jo netop idéen at øge abstraktiuonsniveauet gennem indkapsling

Lad os se på et eksempel hvor vi bruger indkapslede data.

```
// Filnavn = inde1.java
// Demonstration af indkapsling
// og rækkevidde

class indkapslet
{
    private int a,b;     // Private data

    public void input(int x, int y)
    {
        a = x + 1; b = y + 1;   // Placerer x og y i de private gemakker
    }

    public void output()
    {
            System.out.println("x = " + a + " y = " + b);
    }
}

public class inde1
{
    public static void main(String args[])
    {
        int X= 3,Y = 4;
        indkapslet stoette = new indkapslet();

        System.out.println("Program start : X = " + X + " Y = " + Y);
        stoette.input(X,Y);     // Kalder objektets interface
        System.out.println("Efter input   : X = " + X + " Y = " + Y);
        stoette.output();
    }
}
```

```
C:\java inde1
Program start : X = 3 Y = 4
Efter input   : X = 3 Y = 4
x = 4 y = 5
C:\
```

Fig 3.2.2

I dette eksempel er de to variabler `a` og `b` indkapslet i klassen `indkapslet`. Disse to variabler kan ikke ses, eller på nogen måde tilskrives fra andre klasser, ej heller fra inde1. Dette er dog ikke nogen hindring for at få dem skrevet ud, som vi gør det i output metoden. Det er blot en betingelse at den metode der vil have tilgang til private data, er medlem af det samme klasse som de data der ønskes adgang til.

Hvad begreberne privat og public har af betydning, kommer vi tilbage til når vi skal forklare rækkevidde begrebet.

Opgave 3.2.1 Skriv en klasse hvor en overloaded metoder *adder* tager to argumenter af typen int, eller double. Summen af disse skal placeres i klassens private variabel temporaer. En anden metode i klassen skal udskrive indholdet af temporaer.

Opgave 3.2.2 Udvid opgave 3.2.1 således, at der er metoder til de 4 regnearter, men kun én udskrivningsmetode. Det skal af udskrivningsmetoden fremgå hvilken af de 4 regnearter der er benyttet sidst.

Tip: Brug en privat variabel til at opbevare info om den valgte regneart.

3.3 Konstruktøren

Konstruktøren, eller constructoren som den kaldes på engelsk, er skabt til at håndtere alle initialiserings opgaver der hidrører til et objekt. Man kan sige at det er den der konstruerer objektet. Alle objekter har en konstruktør, hvis ikke programmøren laver en vil der automatisk blive oprettet en default konstruktør af Java. Linien:

indkapslet stoette = new indkapslet();

fra Fig 3.2.2 slutter med (), dette skyldes at konstruktøren kaldes. Konstruktøren er en helt almindelig metode der har den egenskab, at den bliver kaldt ved et objekts fødsel, og ellers aldrig. Det er en fejl at kalde en konstruktør i andre sammenhænge.

```
// Filnavn = konstr1.java


class divider
{
        private int godnok,x,y;

        public divider()
        {
                godnok = 0;
        }
        public void input(int a, int b)
        {
                x = a; y = b;
                godnok = 1;
        }
        public void output()
        {
                if (godnok != 0)
                        System.out.println(x/y);
                else
                        System.out.println("Fejl: Indhold udefineret");
                godnok = 0;
        }
}


public class konstr1
{
        public static void main(String args[])
        {
                divider a = new divider();

                a.output();       // Denne fejl fanges
        }
}
```

```
C:\java konstr1
Fejl: Indhold udefineret
C:\
```

Fig 3.3.1

I Fig 3.3.1 bruges konstruktøren til at initiere den private variabel `godnok`. Denne variabel bliver sat til 1 hvis der har været udført en input funktion. Hvis der ikke er noget input, så risikerer vi at komme til at dividere med nul, hvilket vil medføre en fejlsituation. Dette undgår vi ved at sikre os at der er tastet noget ind.

Eksemplet kan være lidt søgt, men det illustrerer konstruktørens evne.

En konstruktør kan overloades, på samme måde som andre metoder. Lad os se på et eksempel hvor vi vil udskrive alle tal i et givet interval. Hvis vi kun opgiver ét argument til konstruktøren, skal den gå ud fra at den skal starte med nul, og slutte med det opgivne tal. Hvis vi opgiver to argumenter til konstruktøren, skal den betragte de to argumenter som yderpunkter i et interval.

```
// Filnavn = interval.java

class runde
{
    private int klar,nedre,oevre;

    public runde()
    {
        klar = 0;
    }
    public runde(int a)
    {
        nedre = 0;
        oevre = a;
        klar  = 1;
    }
    public runde(int a, int b)
    {
        nedre = a;
        oevre = b;
        klar  = 1;
    }
    public void goerdet()
    {
        if (klar != 0)
            for (int i = nedre; i <= oevre; i = i + 1)
                 System.out.println(i);
    }
}

public class interval
{
        public static void main(String args[])
        {
                runde a = new runde(3);
                runde b = new runde(5,9);

                a.goerdet();
                b.goerdet();
        }
}
```

```
C:\java interval
0
1
2
3
5
6
7
8
9

C:\
```

Fig 3.3.2

Klassen runde har 3 konstruktører. Den første sikrer at metoden `goerdet()` ikke får lov til at køre hvis variablerne ikke har fået en kendt værdi: Dette sker ved at sætte `klar = 0`. Den næste konstruktør bliver kaldt hvis objekter bliver født med ét argument, den sidste konstruktør bliver kaldt hvis objektet fødes med to argumenter.

Opgave 3.3.1 Udvid programmet fra Fig 3.3.2 med en konstruktør, således at objektet kan tælle med en trinstørrelse der defineres ved fødslen af objektet.

Opgave 3.3.2 Udvid programmet fra Fig 3.3.2, således at objektet kan tælle baglæns hvis det bliver født til det.

Opgave 3.3.3 Skriv en klasse, hvor konstruktøren initieres med variablen hastighed. Tilføj derefter en metode der tager argumentet minutter. Denne metode skal udskrive hvor langt man kommer på minutter, hvis man kører med den initierede hastighed.

Opgave 3.3.4 Skriv en klasse ved navn tid. Klassen skal have en privat metode der kan regne minutter om til timer og minutter, og timer om til minutter.

Opgave 3.3.5 Udvid 3.3.4 med metoder til at addere to tidspunkter, uanset om disse er opgivet i timer og minutter eller blot minutter.

Opgave 3.3.6 Udvid 3.3.4 med metoder til at subtrahere to tidspunkter, uanset om disse er opgivet i timer og minutter eller blot minutter.

Opgave 3.3.7 Skriv en klasse der gennem et initieret interval, givet gennem konstruktøren, har en metode der finder alle lige tal.

Opgave 3.3.8 Udvid 3.3.7 med en metode `bestem(int tal)`. Metoden skal udskrive alle tilfælde i det initierede interval som er deleligt med tal.

3.4 Nedarvning

Nedarvning et en vigtig element i objektorienteret programmering. Nedarvning betyder, at en klasse kan få egenskaber fra en anden klasse, ved at arve fra den klasse den ønsker egenskaber fra. I Java kan hver klasse nedarve fra én, og kun én klasse, denne klasse (som der nedarves fra) kaldes klassens superklasse. Klassen der nedarver kaldes subklassen. (sub = under) Hvis man bruger begreberne barn og forældre, så er forældre klassen barnet's superklasse, og barnet kan derved arve sin forældre metoder og variabler. (der er kun én forældre i Java)

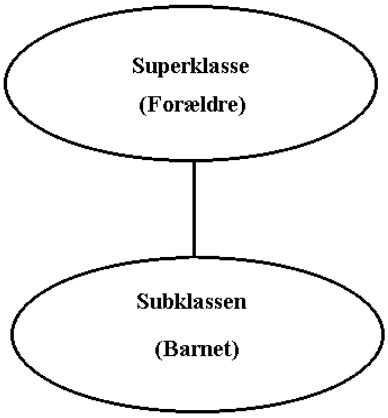


Fig 3.4.1

Fig 3.4.1 viser nedarvning på grafisk form, i praksis kan det se ud som på Fig 3.4.2. Nedarvning foregår altid nedad, aldrig opad. En klasse har altid én, og kun én superklasse, selv om superklassen nedarver fra en anden klasse, så er superklassen altid den man arver fra i direkte linie.

```
// Filnavn = nedarv1.java

class foraeldre
{
    int a,b,c;

    public void Metode1()
    {
      System.out.println(c);
    }
}

class barn extends foraeldre
{
    public void Barnet()
    {
            a = 2; b = 2;
            c = a + b;
            Metode1();
    }
}
```

```
public class nedarv1
{
   public static void main(String args[])
   {
       barn x = new barn();
       x.Barnet();
   }
}
```

```
C:\java nedarv1
4

C:\
```

Fig 3.4.2

Kildetekst på Fig 3.4.2 viser programmet nedarv1.java. `main()` opretter et objekt af typen barn, denne klasse er nedarvet fra klassen `foraeldre`. Derved kan metoden `Barnet()` i klassen barn benytte variablerne `a`, `b` og `c`, som var det dens egne. `Barnet()` kan endvidere benytte metoden `Metode1()` fra superklassen (foraeldre) som er det en lokal metode.

Flere klasser kan nedarve fra den samme klasse, det er faktisk det der gør nedarvning interessant. I Fig 3.4.2 kunne vi udvide kildeteksten med en klasse `soester`, som kunne noget andet end barn, men som har alle egenskaberne fra `foraeldre`. En sådan klasseerklæring kunne se således ud.

```
class soester extends foraeldre
{
     soestsermetode()
     {
          int x = 5, y = 12;

          c = x + y;
          Metode1();
     }
}
```

Fig 3.4.3

Hvis man samtidig satte følgende linier ind i main():

```
soester unge = new soester();

......

unge.soestermetode();
```

Ville der komme et resultat mere ud under kørslen, nemlig summen af 5 og 12 = 17.

Opgave 3.4.1 Skriv et program der har superklassen koeretoej. Denne klasse skal indeholde variabler der kan indeholde vægtafgift og registreringsnummer. Programmet skal envidere indeholde tre subklasser, personvogn, lastbil og bus.

Klassen personbil skal have variabler til at angive hastighed, motorstørrelse og antal hestekræfter.

Klassen lastbil skal indeholde variabler for antal hestekræfter og lastkapacitet.

Klassen bus skal indeholde varibler for antal passagerer, og antal døre.

Programmet skal have metoder til at udskrive alle data.

Tip: Lad konstruktøren initialisere variabler til testformålet.

3.4.1 Klassehiraki

Nedarvning er først interessant når det foregår i stor stil, som skitseret i Fig 3.4.4. Af denne figur fremgår det, at der er en klasse AAAA, denne har nedarvet fra klassen BBB, der har nedarvet fra klassen AA, og denne har igen nedarvet fra klassen A. Dette betyder, at alle metoder og variabler der er erklæret i klassen A, kan benyttes af klassen AAAA.

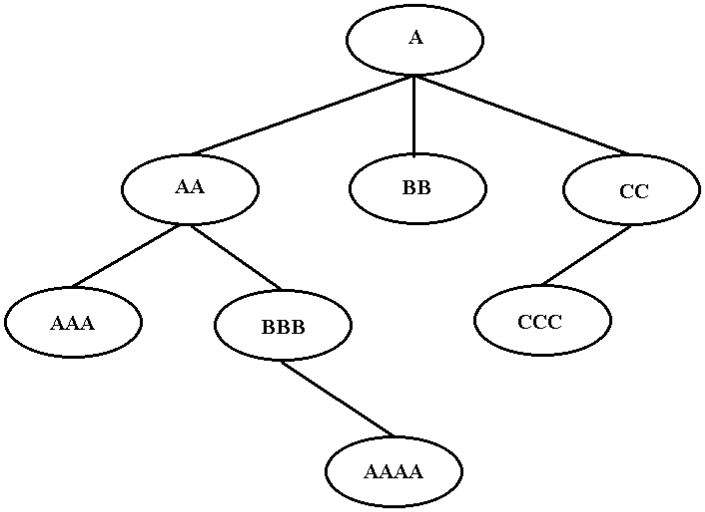


Fig 3.4.4

Hvis klassen AAAA også havde nedarvet fra en , eller flere af de andre klasser, ville der have været tale om multipel nedarvning, dette er ikke tilladt i nuværende versioner af Java.

Opgave 3.4.2 Tegn klassehirakiet for opgave 3.4.1.

3.5 Overskrivning

Når en klasse nedarver fra en anden klasse, kan subklassen se alle de variabler og metoder der er erklæret i superklassen, og superklassens superklasse, o.s.v. Metoder kan overskrives i subklasser, blot ved at programmøren definerer en metode med samme navn og argumentliste som den man ønsker oveskrevet. Variabler kan overskrives ud fra samme regel. Hvis både superklassen og subklassen har defineret en variabel ved navn tst, vil der være tale om to forskellige variabler. En overskrevet variabel kan refereres ved at sætte `super.` foran variabel navnet. Denne teknik er vist i Fig 3.5.1

```
// Filnavn = nedarv2.java

class foraeldre
{
        int a = 2,b = 2,c;

        public void Metode1()
        {
                System.out.println(c);
        }
}

class barn extends foraeldre
{
        public void Barnet()
        {
                int a = 3;

                c = a + b;
                Metode1();
                c = super.a + b;
                Metode1();
        }
}

public class nedarv2
{
       public static void main(String args[])
        {
              barn x = new barn();
              x.Barnet();
        }
}
```

```
C:\java nedarv2
5
4

C:\
```

Fig 3.5.1

Metoden `barn` overskriver superklassens variabel `a`, men kan benytte den ved at kalde den `super.a`.

Denne teknik kan kun anvendes ét trin opad. Der er ingen direkte måde til at få adgang til superklassens, superklasse´s variabler og metoder der er overskrevet af superklassen.

Metoder kan overskrives efter samme princip som variabler. Hvis Metode1() i Fig 3.5.1 blev overskrevet i subklassen, kunne superklassens udgave af Metode1() kaldes ved at skrive.

```
super.Metode1();
```

Dette kan se ud som vist på Fig 3.5.2

```
// Filnavn = nedarv3.java

class foraeldre {
        int a = 2,b = 2,c;

        public void Metode1()
        {
                System.out.println(c);
        }
}

class barn extends foraeldre {
        // Her overskrives superklassens metode
        public void Metode1()
        {
            System.out.println("c = " + c);
        }

        public void Barnet()
        {
            // Her overskrives superklassens variabel
            int a = 3;

            c = a + b;
            super.Metode1();
            Metode1();
            c = super.a + b;
            super.Metode1();
            Metode1();
        }
}

public class nedarv3 {
       public static void main(String args[])
        {
              barn x = new barn();
              x.Barnet();
        }
}
```

```
C:\java nedarv3
5
c = 5
4
c = 4
C:\
```

Fig 3.5.2

Med samme teknik kan man også kalde superklassens konstruktør. På Fig 3.5.2 ville et kald til superklassens konstruktør se således ud:

```
super();
```

Der må ikke indgå klassenavn i kaldet til superklassens konstruktør.

Hvis konstruktøren er overloaded, som i Fig 3.5.2 kunne kald til de tre konstruktører se ud på én af følgende 3 måder, forudsat at vi befandt os i en subklasses konstruktør.

```
int x,y;

super();
```

```
int x,y;

super(x);
```

```
int x,y;

super(x,y);
```

Det vil være en fejl at forsøge at kalde superklassens konstruktør to gange, da der altid bruges en, og kun en konstruktør til at konstruere et objekt. Kaldet til superklassens konstruktør skal være det første der foretages, efter evt. variabel erklæringer i subklassens konstruktør. Alle andre forsøg på at kalde superklassens konstruktør skal medføre en kompilerfejl.

Opgaver 3.5.1

Skriv et program der nedarver fra en klasse med én konstruktør. Undersøg i hvilken rækkefølge konstruktørerne bliver kaldt. (Bliver subklassens konstruktør udført før superklassens konstruktør, eller omvendt?)

3.6 Abstrakte klasser og metoder

I objektorienteret programmering kan man komme ud for at skulle konstruere klasser der er så generelle, at de er uanvendelige til noget specifikt. Med andre ord, en sådan klasse bør der ikke oprettes et objekt af direkte, men nedarvning til andre klasser vil være naturlig. En sådan klasse kaldes en abstrakt klasse.

Defenition: En abstrakt klasse er en klasse der ikke må kunne skabes et tilfælde (objekt) af.

En abstrakt klasse kan defineres i Java, med det reserverede ord `abstract.` En klasse der er defineret som abstrakt, kan der ikke oprettes et tilfælde af. Hvis man alligevel forsøger, vil kompileren melde noget i retning af:

class xxxxx is an abstract class, it can't be instansiated.

En abstrakt klasse defenition kan se som vist i Fig 3.6.1.

```
abstract class minklasse
{
     ...
}
```

Fig 3.6.1

Abstrakte klasser vil typisk være placeret højt i klassethirakiet, fordi de udtrykker noget meget generelt.

Abstrakte metoder kan bruges i abstrakte klasser, eller klasser der er nedarvet af abstrakte klasser, til at tvinge en bruger (programmør) af en abstrakt klasse, til at implementere specifikke metoder. En abstrakt metode er en metode uden kode, selve koden skal implementeres i de nedarvede klasser, på samme måde som man overskriver en metode.

En abstrakt metode kan se således ud, bemærk at der er ingen tuborg parenteser, og linien afsluttes med et semikolon.

```
abstract void min_abstrakte();
```

Dette betyder, at alle ikke abstrakte klasser der nedarver fra klassen, skal oprette en metode ved navn min_abstrakte, og metoden må ikke returnere nogen værdi.
Lad os kigge på et eksempel:

```
// Filnavn = abstrakt1.java

abstract class hus
{
    int aabninger, doere, vinduer;

    abstract int antal_aabninger();
}

class villa extends hus
{
    villa(int a, int b)
    {
        doere = a;
        vinduer = b;
    }

    // Her implementeres den abstrakte metode
    int antal_aabninger()
    {
        return doere + vinduer;
    }
}


public class abstrakt1
{
    public static void main(String argv[])
    {
        villa hjem = new villa(3,4);

        System.out.println(hjem.antal_aabninger());
    }
}
```

```
C:\java abstrakt1
7

C:\
```

Fig 3.6.2

Da klassen `villa` ikke er erklæret abstrakt, skal den implementere den abstrakte metode `antal_aabninger()`. Med denne teknik tvinger programmøren af klassen `hus`, brugere af klassen til at gøre den færdig i hans ånd. Skulle brugeren glemme at implementere evt. abstrakte metoder, vil han blive erindret af kompileren, der vil nægte at kompilere kildeteksten, indtil alle abstrakte metoder er implementeret.

Opgave 3.6.1 Omskriv programmet fra opgave 3.4.1, således at klassen `koeretoej` bliver erklæret som en abstrakt klasse.

3.7 Adgangskontrol

Man kan styre hvem der må have adgang til hvad i et objekt, ved at benytte access specifiers. I Java findes der fire access specifiers, private, protected, public og package.

private

Dette er den mest restriktive specifier. Hvis en metode eller variabel erklæres som private, så kan den kun ses af metoder der er medlem af klassen. Private variabler og metoder kan ikke ses i nedarvede klasser.

protected

Hvis en metode eller variabel er erklæret som protected betyder det, at den kun er tilgængelig fra medlemmer af klassen, medlemmer i klasser der arver fra klassen, og i klasser der er placeret i samme pakke som variablen/metoden. (Vi kommer senere tilbage til pakker)

public

Dette betyder, at alle der kan lave et tilfælde (objekt) af klassen, også kan bruge denne metode/variable. Det er ideelt set forkert at erklære variabler som public, men man gør det dog ofte, og slipper glimrende fra det, men erklær kun variabler som public med omtanke.

package

Dette er det accessniveau der bliver sat, hvis man ikke definerer noget. (Default) Niveauet betyder, at kun klasser i samme pakke har adgang til variablen/metoden.

Med disse fire access specifiers kan man udføre indkapsling. Ved at kigge på Fig 3.2.1 kan man sige, at de metoder der kan "ses" udenfor klassen, er klassens interface. Disse metoder er typisk erklæret som `public`. De indkapslede metoder vil typisk være erklæret som `private`. De to mellemliggende specifiers bliver lidt nemmere at forstå når vi kommer lidt videre.

3.7.1 Et eksempel

I Fig 3.7.1 er en kildetekst der undtagelsesvis ikke kan kompiles. Dette er valgt fordi det kan give et eksempel på hvorledes access specifierne virker. De indsatte kommentarer bør være tilstrækkelig forklaring.

```
// Filnavn access.java

// Eksempel til at visualisere brugen af access specifiers
// Kan IKKE kompiles

class superklassen
{
    public  int a,b;
    private int c,d;

    private void print(int x)
    {
        System.out.println(x);
    }

    protected void print_smukt(int x)
    {
        System.out.println("Summen er : " + x);
    }

    public void adder(int x, int y)
    {
        print(x+y);
        print_smukt(x + y);
    }
}

class nedarv extends superklassen
{
    nedarv()
    {
        a = 1; b = 2;
        c = 3;                          // Illegalt, c er private i superklassen

        print(a + b);                   // Illegalt, print(int) er private
        print_smukt(a + b);
    }
}

public class access
{
    public static void main(String args[])
    {
        superklassen a = new superklassen();
        nedarv      b = new nedarv();

        a.adder(1,3);
        a.print(2 + 2);                 // Illegalt, print er private
        a.print_smukt(2 + 2);           // Tilladt fordi vi er i samme pakke

        b.adder(1,3);
        b.print(2 + 2);                 // Illegalt, print er private
        b.print_smukt(2 + 2);           // Tilladt fordi vi er i samme pakke
    }
}
```

Fig 3.7.1

3.8 opgaver

Opgave 3.8.1 Skriv et program med den abstrakte superklasse ejendom. Tre nedarvede klasser villa, butik og etage skal have følgende egenskaber.

villa

Konstruktøren skal modtage antal rum, antal beboede kvadratmeter og pris.

butik

Konstruktøren skal modtage beboelsesareal, butiksareal samt pris.

etage

Konstruktøren skal modtage beboelsesareal, butiskareal, antal_etager og pris.

Alle klasserne skal have en metode til at udskrive pris per kvadratmeter.

3.9 Interface

Et interface er den yderste konsekvens af en abstrakt klasse. Et interface er en abstract klasse der kun må indeholde konstanter og abstrakte metoder.

Man kan nedarve fra et interface, men man kan ikke skabe et objekt af et interface. Fig 3.9.1 viser et eksempel på hvordan et interface fungerer.

```
// Filnavn = intfac.java

interface do_it {
    int MAX = 99;

    void print();
}

class do_it_again implements do_it {
    public void print()     {
        System.out.println(MAX);
    }
}

public class intfac {
    static void tst() {
        do_it_again a = new do_it_again();

        a.print();
    }

    public static void main(String arg[])     {
        tst();
    }
}
C:\java intfac
99

C:\
```

Fig 3.9.1

Et interface kan altså nedarves med det reserverede ord `implements`. Man kan nedarve mere end et interface, blot ved at adskille interface navnene med et komma. Dette kan forveksles med begrebet multipel nedarvning, men det er der ikke tale om, da et interface kun kan bestå af abstrakte metoder.

I Fig 3.9.1 kan ingen ændre på værdien i `MAX`, dette skyldes at den eksplicit er erklæret som `public, static` og `final`. (Vi kommer tilbage til begreberne static og final) På denne måde bliver alle "variabler" erklæret i et interface til konstanter, fordi de ikke kan ændres

4 Pakker

Pakker benyttes til at organisere klasser på en praktisk måde. En pakke er en samling af tektsfiler, samlet i et bibliotek, med pakkens navn som bibliotekets navn. Hver fil udføres som en normal sourcefil, indeholdende en samling af klasser. Det er god programmeringsskik, at lade en pakke indeholde relaterede ting. Bemærk i øvrigt, at der kun må være en klasse i hver fil der erklæres som public.

4.1 Sådan fremstilles en pakke

I toppen af en pakke, skal det fremgå at der er tale om en pakke, samt hvilket navn pakken har. Dette gøres med det reserverede ord package. Fig 4.1.1 viser to filer i pakken ved navn pakke. Disse to filer skal placeres i et bibliotek ved navn pakke, og dette bibliotek skal være placeret lige under arbejdsbiblioteket, eller i en særlig filstruktur. (Hvis pakker ønskes placeret andre steder, kan dette gøres ved at ændre systemvariablen CLASSPATH. Se kompilerens dokumentation for yderligere detaljer.)

```
// Filnavn = pakke1
// Filen skal placeret i biblioteket pakke, et sted
// synligt i CLASSPATH
// Se din kompilers README fil for  yderligere
// info om brug af CLASSPATH

package pakke; //Identificerer tilhørsforhold

// Bemærk: klasse1 må ikke erklæres public
class klasse1
{
        public void show()
        {
                System.out.println("Klasse 1 i pakke1");
        }
}

// pakke klassen skal erklæres public
public class pakke1
{
        public void show()
        {
                System.out.println("Show i pakke1");
        }
}
```

Fig 4.1.1a

```
// Filnavn = pakke2.java
// Denne fil skal have navnet pakke2.java
// Filen skal placeret i biblioteket pakke, et sted
```

```
// synligt i CLASSPATH

package pakke; //Identificerer tilhørsforhold

class klasse2
{
        public void show()
        {
                System.out.println("Klasse 2 i pakke2");
        }
}

public class pakke2
{
        public void show()
        {
                System.out.println("Show i pakke2");
        }
}
```

Fig 4.1.1b

Bortset fra at begge filer starter med “package pakkenavn;” er der ikke nogen forskel på disse filer, og dem vi har lavet hidtil. Filerne kompileres på normal vis. Når .class filerne er fremstillet, er vi klar til at bruge dem i et normalt program.

Fig 4.1.2 viser et program der gør brug af de netop fremstillede pakker. For at få fat i de klasser der er inde i filerne, er vi nød til først at importere dem. Dette gøres med det reserverede ord import.

```
// Filnavn = useit.java
// Dette program importerer fra pakken pakke.
// Pakkens class filer skal ligge kompileret under
// denne fils arbejdsbiblotek, i et biblotek
// ved navn pakke, eller det skal på anden
// måde være synligt i CLASSPATH

import pakke.*; // Importer hele pakken.

class minklasse
{
        minklasse()
        {
                pakke1  a = new pakke1();
                pakke2 b = new pakke2();
//              klasse1 c = new klasse1();  !!! Ulovligt
//              klasse1 kan kun ses af pakkens medlemmer

                a.show();
                b.show();
        }
}

public class useit
{

        public static void main(String arg[])
        {
                minklasse ex = new minklasse();
        }
}
```

```
C:\java useit
Show i pakke1
Show i pakke2

C:\
```

Fig 4.1.2

Ved at skrive:

```
import pakke.*;
```

importerer vi, og får dermed adgang til alle klasser i pakken ved navn pakke. Havde vi i stedet skrevet:

```
import pakke.pakke1;
```

Havde vi fået en kompilerfejl i den linie hvor vi forsøger at oprette et tilfælde af `pakke2`. Dette skyldes at `pakke2` ikke er i filen pakke1, men i filen pakke2. Vi kunne i stedet have skrevet:

```
import pakke.pakke1;
import pakke.pakke2;
```

Det vil virke, men er nok lidt mere besværligt end at tage begge pakker i en omgang med `import pakke.*;` Det er fristende, men forkert at skrive

```
import pakke.p*;
```

Træerne vokser desværre ikke helt ind i himlen

Det der står mellem import og det første punktum, er altså et biblioteknavn. Det efterfølgende har hidtil været et filnavn, men det kunne også have været navnet på et bibliotek, placeret under det første bibliotek, i så fald skal der et punktum mere med, for at komme frem til filnavnet. Punktummet fungerer altså stort set som en backslash (\) i DOS, eller en slash (/) i UNIX®.

Opgave 4.1.1 Placer skudårs metoden fra 2.5.2 i en pakke, og test funktionaliteten ved at importere pakken i et testprogram.

Opgave 4.1.2 Opret en pakke ved navn matematik. Pakken skal indeholde fire klasser, en for hver af de grundliggende regnearter. I hver klasse skal der kunne regnes med int og float.(metoderne skal overloades)

Opgave 4.1.3 Udvid matematik pakken med en klasse der kan beregne x^y.

4.2 Standardpakker

Der følger en del standardpakker med når man anskaffer sig en Java kompiler. De mest interessante er kort beskrevet i det følgende. For fuldstændig information, se under API i filen jdkdoc der er en Windows help fil der følger med JDK'et.

java.lang

Dette er en standardpakke, der altid importeres. Det er en fejl at importere den manuelt. Udvalgte dele af pakken bliver gennemgået i næste afsnit.

java.io

Denne klasse indeholder nogle input og output streams, til at udskrive og hente data, fra eksempelvis keyboard, skærm og fil.

java.util

Inderholder nogle praktiske metoder til datamanipulation.

java.awt

awt betyder Abstract Windows Toolkit. Denne pakke indeholder metoder til at arbejde med vinduer. Vi kommer tilbage til awt'et i kapitel 9.

java.applet

Indeholder metoder og klasser til at arbejde med appletter. Alle appletter skal nedarve fra klassen Applet, der er placeret i denne pakke. I pakken er også faciliteter til at håndtere audio og video.

4.3 java.lang

Pakken indeholder på nuværende tidspunkt 2 interface (cloneable og runable) 24 klasser, samt et antal exceptions og fejlhåndtering. De sidste to ting kommer vi tilbage til senere, men vi vil starte med klassen Object i pakken java.lang.

4.3.1 Object

Klassen er interessant, fordi det er den øverste klasse i Java's klassehiraki. Hvis en klasse ikke specificerer at den nedarver fra en anden klasse, så arver den per automatik fra klassen Object. Alle klasser nedarver på en eller anden måde fra klassen Object.

4.3.2 String

Vi har benyttet String lige siden vi startede, uden at tænke nærmere over det. I mange programmeringssprog er en string blot en samling af bogstaver. Det er ikke tilfældet i Java. String er en klasse der kan instansieres med mange forskellige konstruktører. De to konstruktører vi vil kigge på er:

```
String()
String(String)
```

Fig 4.3.2.1 viser et eksempel på brug af den første konstruktør:

```
// Filnavn = streng1.java

public class streng1
{
        public static void main(String arg[])
        {
                String minStr = new String();

                minStr = "Hallo der";
                System.out.println(minStr);
        }
}
```

```
C:\java streng1
Hallo der

C:\
```

Fig 4.3.2.1

For gamle Basic, Pascal, C og C++ programmører ser det afgjort anderledes ud, end de er vandt til. Men for os er det jo nemt nok, vi ved jo at `minStr` ikke er en variabel, men et objekt af typen String.

En anden måde at få tilskrevet `minStr` er ved at benytte den konstruktør der modtager en String, det kan se ud som på Fig 4.3.2.2

```
// Filnavn = streng2.java

public class streng2
{
        public static void main(String arg[])
        {
                String minStr = new String("Hallo der");

                System.out.println(minStr);
        }
}
```

```
C:\java streng2
Hallo der

C:\
```

Fig 4.3.2.2

Vi kan udvidde det med noget vi har prøvet før, nemlig at addere en streng til en streng. Det har vi jo tidligere gjort i argumentet til en printmetode.

```
// Filnavn = streng3.java

public class streng3
{
        public static void main(String arg[])
        {
                String minStr = new String("Hallo ");

                minStr = minStr + "der";
                System.out.println(minStr);
        }
}
```

```
C:\java streng3
Hallo der

C:\
```

Fig 4.3.2.3

I klassen String er der en metode til at bestemme længden af en streng. Metoden hedder length() og kan benyttes på denne måde. Se Fig 4.3.2.4.

```
// Filnavn = streng4.java

public class streng4
{
        public static void main(String arg[])
        {
                int a;
                String minStr = new
String("Teststreng");

                a = minStr.length();
                System.out.println(a);
        }
}
```

```
C:\java streng4
10

C:\
```

Fig 4.3.2.4

I linien:

```
a = minStr.length();
```

tilskriver vi variablen `a` med værdien af længden af `minStr`. Det må betyde, at metoden length returner en `int`.

Vi kunne have sparet en linie i Fig 4.3.2.4, og i stedet have skrevet:

```
System.out.println(minStr.length());
```

En af de andre metoder i String klassen er `compareTo`. Denne metode kan vi benytte til at undersøge om to strenge er ens, det kan se ud som vist i Fig 4.3.2.5.

```
// Filnavn = streng5.java

public class streng5
{
    public static void main(String arg[])
    {
        String minStr1 = new String("Testord");

        if (minStr1.compareTo(arg[0]) == 0)
                System.out.println("De er ens");
        else
                System.out.println("De er IKKE ens");
        }
}
```

```
C:\java streng5 Testord
De er ens

C:\
```

Fig 4.3.2.5

Opgave 4.3.2.1 Skriv et program der ændrer alle bogstaverne i en teststreng til store bogstaver.
Tip: Undersøg java.lang pakken, og kig efter String klassen. Disse informationer findes i javadokumentationen, der kan hentes fra www.javasoft.com.

Opgave 4.3.2.2 Skriv et program der ændrer alle bogstaverne i en teststreng til små bogstaver.

Opgave 4.3.2.3 Skriv et program der undersøger om der er to ens argumenter på kommandolinien.

4.4 Array's

Array's benyttes til at have serier af den samme type. Et Array skal have en defineret størrelse. En array kan erklæres på denne måde:

```
int minArray[] = new int[10];
```

Eller således:

```
int[] minArray = new int[10];
```

Dette skaber et objekt af typen `int` indeholdende 10 int's, fra posision 0 til 9. De enkelte elementer i arrayet kan da manipuleres, som vist i Fig 4.4.1.

```
// Filnavn array1.java

public class array1
{
    public static void main(String arg[])
    {
            int i;
            int minArray[] = new int[10];

            for (i = 0; i < 10; i = i + 1)
                minArray[i] = i;
            for (i = 9; i >= 0; i = i - 1)
               System.out.println(minArray[i]);
        }
}
```

```
C:\java array1
9
8
7
6
5
4
3
2
1
0

C:\
```

Fig 4.4.1

Der kan naturligvis skabes array's af alle typer, også objekter. Men netop når man skaber et array af objekter, skal man huske på, at det er ikke nok at oprette en array med new. Hvis array'et indeholder objekter, skal der oprettes et tilfælde af hvert objekt i arrayet. Det kan se således ud:

```
String str[] = new String[10];

for (int i = 0; i < 10; i = i + 1)
     str[i] = new String();
```

Hvis man undlader det, får man en NullPointerException under runtime.

4.5 Opgaver

Opgave 4.5.1 Skriv et program der placerer længden af de enkelte argumenter fra kommandolinien i seperate dele af et array af typen int.
Tip: java.lang.length()

Opgave 4.5.2 Udvid 4.5.1 med en metode til at bestemme den gennemsnitlige længde af de ord der fremgår af kommandolinien.

4.6 Et praktisk eksempel med pakker

Vi vil slutte dette kapitel af med at konstruere en pakke der kan bruges til noget så irriterende som datoaretmetik. Det vil sige, vi vil fremstille en pakke der bl.a. kan håndtere hvor lang afstand der er mellem to datoer. Senere vil vi måske udvidde den til også at kunne lægge to datoer sammen, finde mærkedage, helligdage m.m. Men vi starter stille og roligt med at opbygge en klasse ved navn kal1. Vi bestemmer allerede nu, at navnet på vores nye pakke skal være kalender. Hermed kan vi fastslå, at vores kildetekst må blive bygget op omkring følgende fil, der skal være placeret i et bibliotek ved navn kalender.

```
// Filnavn = kal1.java
package kalender;

class kal1
{
}
```

Nu skal vi bare putte noget fornuftigt i klassen. Vi kan jo starte med at genbruge skudårs metoden fra 2.5.2. Så skal vi faktisk bare have en metode der kan regne en dato på formen år/måned/dag om til den jullianske dato. (Den jullianske dato er dagens nummer i året. Der er 366 dage i et skudår, ergo kan den jullianske dato aldrig blive større end 366.

Når vi kan finde dagnummeret, så har vi næsten løst opgaven. Lad os se på hvordan det kan gøres.

```
// Filnavn = kal1.java

package  kalender;

public class kal1
{
    byte dageImdr[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};

    // Returnerer true hvis aar er et skudaar, ellers returneres false
    boolean skudaar(int aar)
    {
        if ( (aar % 400 == 0) || (aar % 4 == 0 && aar % 100 != 0))
                return true;
        else return false;
    }

    // Modtager årstal, månedsnummer og dag i måned
    // Returner dagnummeret (Den julianske dato)
    public int julliansk_dag(int aar, int mdr, int dag)
    {
        int i;

        for (i = 1; i < mdr; i++)
            dag += dageImdr[i];
        if (mdr > 2 && skudaar(aar))
            dag++;
        return dag;
    }
```

```
}
```
Intet output

Fig 4.6.1

Hvis du sidder og undrer dig over at der ikke er nogen main metode i pakken, så er det fordi der ikke er noget at bruge en main metode til. Men vi skal nok komme til en main metode.

Programmet starter med linien:

```
package  kalender;
```

Det betyder at alt i denne fil tilhører pakken ved navn kalender. Alle filer tilhører enten en pakke angivet på denne måde, eller det der kaldes default pakken. Hidtil har vi kun arbejdet med default pakken.

Det første der sker inde i klassen kal1 er, at der initieres et array af typen `byte`, det er tretten elementer langt, og det indeholder nogle praktiske tal. Den snedige læser kan se, at posisionsnummeret i arrayet er lig med et månedsnummer, hvorefter indholdet i posisionen er antal dage i den måned. (januar måned er måned nr. 1 og den har 31 dage)

Årsagen til at der er valgt `byte` og ikke `int` som type, er blot forfatterens trang til at klemme alt muligt ind på så lille hukommelsesplads som muligt.

Metoden `skudaar(int aar)` kender vi jo, så den er der ingen grund til at komme nærmere ind på, vi genbruger bare noget kode vi har fremstillet tidligere.

Metoden `public int julliansk_dag(int aar, int mdr, int dag)` er straks mere interessant. Metoden tager en dato på formen år/måned/dag og returnerer den jullianske dag. Det gøres ved, måned for måned, at summere antallet af dage i den givne måned. Antallet af dage i måned `i` fremgår jo netop af det i'de element i `dageImdr`. Når det er afsluttet, testes om der er tale om et skudår, og om måneden er større end februar. Hvis det er tilfældet skal skudagen lægges til antallet af dage.

Vi er ikke færdige med opgaven endnu, men det lysner forude. Inden vi går videre skal vi lige have gemt filen, og kompileret den. Hvis filstrukturen ser således ud

D:\JDKversionnr\kode\kap4

og du er ved at arbejde med kapitel 4 opgaverne i biblioteket ved navn kap4. Så vil jeg anbefale, at du opretter et bibliotek under kap4, ved navn kalender. Det kan se således ud

D:\JDKversionnr\kode\kap4\kalender

I det bibliotek skal alle classfiler til kalenderpakken forefindes. Kildeteksterne behøves ikke ligge i det bibliotek, men for at gøre det nemt bør kildeteksten gemmes således:

D:\JDKversionnr\kode\kap4\kalender\kal1.java

Herefter kan du skifte til kap4 biblioteket, ikke kalender biblioteket, og kompilere kal1.java ved at skrive

javac kalender\kal1.java

herved sikrer du dig at kompileren kan finde ud af det hele. Du kan godt skifte ned i kalender biblioteket og kompile kildeteksten for nuværende, men når vi udvidder pakken vil der opstå problemer.

Når kal1 er kompilet kan vi prøve den del af pakken af med programmet testkal1.

```
// Filnavn = testkal1.java
import kalender.*;

public class testkal1
{
    public static void main(String arg[])
    {
        kal1 a = new kal1();

        System.out.println(a.julliansk_dag(2000,12,31));
    }
}
```

```
C:\java testkal1
366

c:\
```

Fig 4.6.2

Programmet starter med at importere alle klasser i kalender pakken, derefter oprettes et objekt af kal1 klassen, og `julliansk_dag` metoden testes på enkleste vis, vi er jo bare ved at kontrollere om vores metode virker. Som det fremgår af output kan den klare år 2000 testen.

Så mangler vi bare at lave en metode der kan finde afstanden mellem 2 datoer. Afstanden mellem to datoer må være differensen i deres jullianske datoer, plus differensen i antal år, med passende hensynstagen til skudår. Dette er ganske præcist formuleret i metoden dato_diff i klassen afstand:

```
// Filnavn = afstand.java
// Beregner afstanden mellem to datoer i dage

package kalender;

public class afstand
{
    public int dato_diff(int aar1, int mdr1, int dag1,
                         int aar2, int mdr2, int dag2)
    {
       int diffDage,i;
       kal1 a = new kal1();


        diffDage = a.julliansk_dag(aar1, mdr1, dag1) -
                   a.julliansk_dag(aar2, mdr2, dag2);
        for (i = aar2; i < aar1; i++)
            if (a.skudaar(i))
                diffDage += 366;
            else diffDage += 365;
        return diffDage;
    }
}
```

Ingen output

Fig 4.6.3

`dato_diff` metoden starter med at fastslå differensen i antal dage mellem de to datoer. Differensen kan udemærket være negativ, det betyder jo blot at der skal lægges et negativt tal til et andet et par linier senere.

Når afstanden i dage er kendt, skal antallet af år mellem datoerne fastslås. Det klares ved et løbe gennem en løkke, fra aar2, til aar1. Hvis det angivne år er et skudår, lægges der 366 til `diffDage`, ellers lægges der blot 365 til.

Det er værd at bemærke, at vi på ingen måde bekymrer os om hvordan vi finder ud af om det er et skudår eller ej. Den bekymringen har vi jo lagt bag os for længe siden. På dette tidspunkt abstraherer vi fra den problematik, vi befinder os på et højere abstraktionsniveau.

For at runde programmeringen af vil vi lige teste om vores pakke fungerer korrekt, det gøres med testkal2 klassen.

```
// Filnavn = testkal2.java
import kalender.*;


public class testkal2
{
    public static void main(String arg[])
    {
        afstand a = new afstand();

        // Beregn aftanden mellem d. 3/1 2001 og 3/1 1999
        System.out.println(a.dato_diff(2001,1,3,1999,1,3));
        // Her er ingen år 2000 problemer :)
    }
}
```

C:\java testkal2
731

c:\

Fig 4.6.2

De to testmetoder vi har anvendt i dette eksempel viser en god måde at teste sig frem på mens man udvikler. Når man laver små programmer til at teste kode med, skal man passe på ikke at teste for detaljeret, så drukner man i tester. Det er dog oplagt at teste alle metoder, gerne en af gangen. Hvis en metode er afhængig af en anden, så start med at teste den anden, så du ikke sider og kommer i tvivl om hvor fejlen er, når du tester den første metode.

Opgave 4.6.1 Udvid kalender pakken med en metode der kan addere to datoer.

Opgave 4.6.2 Udvid kalender pakken med en metode der kan omregne fra julliansk dato, til måned og dag i måned.

Opgave 4.6.3 Diskuter om følgende klasse har nogen berettigelse i kalender klassen.

```
public class datoklasse
{
     int aar;
     byte mdr;
     byte dag;

     dataklasse(int aar, int mdr, int dag)
     {
          this.aar = aar;
          this.mdr = mdr;
          this.dag = dag;
     }
}
```

5 Exceptions

Exceptions betyder undtagelser, exceptions benyttes altså til at håndtere undtagelser. Undtagelser er når der opstår noget uventet, og uventede begivenheder skyldes en fejl, exception handling er altså fejlhåndtering.

Vi vil starte vejen ud i fejlhåndtering, ved at kigge på et eksempel der kræver fejlhåndtering for at kunne kompileres.

5.1 Input fra tastatur.

Lad os kigge på Fig 5.1.1 for at se et program der henter et input fra tastaturet, og derefter skriver det ud igen i omvendt rækkefølje.

```
// Filnavn = input1.java

// Henter karakterer fra tastatur, indtil der trykkes på
// ENTER (ASCII 13). Programmet kan fejle under runtime,
fordi
// der ikke er kontrol af overløb

public class input1
{
    public static void main(String arg[])
        throws java.io.IOException
    {
        int i,j;
        char stak[] = new char[10];

        i = 0;
        while ( (stak[i] = (char) System.in.read() ) != 13)
                i = i + 1;
        for ( j = i; j >= 0; j = j - 1)
                System.out.print(stak[j]);
    }
}
```

```
C:\java input1
abc↵
cba
C:\
```

Fig 5.1.1

Der er flere nye ting vi skal kigge på her. For det første er der linien:

```
throws java.io.IOException
```

Dette er en specificering af, at objektet kan kaste (throw) et exception af typen IOException. Denne skal så være defineret andet sted. (I dette tilfælde i java.io)

Der er en anden og mere skjult fejlrisiko. Hvad nu hvis vi indtaster mere end 10 tegn inden vi trykker på enter. I så fald vil array'et forsøge at skrive ud over sin længde, dette er ulovligt og umuligt i Java, altså opstår der en fejl. Inden vi begynder at fange denne fejl, vil vi lige gennemgå et par nye begreber der er dukket op i dette eksempel.

Bemærk linien

```
while ( (stak[i] = (char) System.in.read() ) != 13)
        ^                                   ^
```

Dette betyder, at vi henter en et tegn fra tastaturet med metoden `System.in.read()`, returværdien af denne metode vil vi gerne have lagt på det i'de sted i stak, men der er et par problemer.

For det første er stak af typen `char`, og `System.out.read()` aflever en `int`. Derfor må vi lave en typekonvertering, også kaldet en cast. Dette er gjort ved at placere `(char)` mellem outputtet og lighedstegnet. Dette betyder, at det der afleveres fra højre side af lighedstegnet, konverteres til en `char` før det tilskrives det i'de element i stak.

I linien ovenfor er der markeret en start og en slut parentes. Hvis disse parenteser blev udeladt, ville der opstå en fejl under kompileringen, da kompileren ville opfatte udtrykket således.

```
while ( stak[i] = (char)  (System.in.read()  != 13) )
                          ^                        ^
```

`(System.in.read()  != 13)` ville aflevere værdien boolsk false hvis der bliver trykket på andet end enter tasten, og true hvis der bliver trykket på enter tasten. Det kan (vil) kompileren ikke konvertere til en char.

Der er stadig en fejlmulighed. Hvis brugeren taster mere end 10 tegn før han trykker på enter, fremkommer en `java.lang.ArrayIndexOutOfBoundsException`. Det vil vi rette i næste afsnit.

Årsagen til vi tester op mod tallet 13 i Fig 5.1.1 er, at tasten enter har talværdien 13 inde i computeren. Alle tegn er repræsenteret inde i computeren med tal, det er årsagen til, at `System.in.read()` metoden returnerer et `int`, og ikke en `char`.

Programmet chartest på Fig 5.1.2 viser et eksempel på hvordan man kan udskrive en tegntabel.

```
// Filnavn chartest.java
// Dette program demonstrerer
// tegntabelens tal værdier

public class chartest
{
    public static void main(String arg[])
    {
        int i;

        for (i = (int) 'A'; i <= (int) 'Z'; i++)
        {
            System.out.print((char) i + " = " + i + " : ");
            if ( (i % 5) == 0)    // Så skal der laves et linieskift
                System.out.println();
        }
    }
}
```

```
C:\java chartest
A = 65 :
B = 66 : C = 67 : D = 68 : E = 69 : F = 70 :
G = 71 : H = 72 : I = 73 : J = 74 : K = 75 :
L = 76 : M = 77 : N = 78 : O = 79 : P = 80 :
Q = 81 : R = 82 : S = 83 : T = 84 : U = 85 :
V = 86 : W = 87 : X = 88 : Y = 89 : Z = 90 :

C:\
```

Fig 5.1.2

Det bringer jo et par nye ting frem i lyset, for det første er der linien:

```
for (i = (int) 'A'; i <= (int) 'Z'; i++)
```

Der jo ser lidt besynderlig ud. Men der står jo "bare", at variablen `i` skal starte med at have talværdien af bogstavet A. Det vises ved at sætte A ind i et par apostrofer, hvorved A bliver en såkaldt karekterkonstant, derefter udføres en cast, fra `char` til `int`. Som det kan ses på udskriften er bogstavet A repræsenteret ved tallet 65, B er repræsenteret ved tallet 66 o.s.v.

Linien:

```
System.out.print((char) i + " = " + i + " : ");
```

starter med at udskrive heltallet i variablen `i` som en `char`, hvilket bliver til et tegn på skærmen. Derefter udskrives et lighedstegn, efterfulgt af i's talværdi. Til sidst sættes et kolon på skærmen, for bedre at kunne se output.

Den sidste linie der kan virke lidt anderledes er:

```
if ( (i % 5) == 0)
```

Men der står jo bare: Hvis tallet fem går op i variablen `i` et lige antal gange, så skal der ske noget. I dette tilfælde skifter vi linie, for ikke at have mere end fem tegn per linie.

Opgave 5.1.1 Skriv et program der udskriver tegntabellen for de små bogstaver

Opgave 5.1.2 Skriv et program der udskriver tegntabellen for tallende nul til og med 9.

Opgave 5.1.3 Tilpas programmet input1, således at de indtastede tegns karekterværdi udskrives.

Opgave 5.1.4 Skriv et program der kryptograferer et input efter Cæsar koden, og udskriver den kryptograferede del.
Tip: Cæsar koden konverterer et a til et b, et b til et c, et z til a o.s.v.

Opgave 5.1.5 Udvid 5.1.4, således at Cæsar koden udviddes med et offset.
Tip: Offset = 2 medfører: a = c, b = d, Offset= 3 medfører a = d, b = e, z = b o.s.v.

Opgave 5.1.6 Skriv et program der givet et Cæsar kryptograferet input med ukendt offset, afprøver alle mulige kodekombinationer (offset's), og skriver dem på skærmen, således at brugeren kan vurdere om koden er knækket.

5.2 At kaste (Throw)

I Fig 5.1.1 tillod vi Java at kaste et objekt blot ved at specificere det. Nu vil vi selv prøve at fange fejlen, før Java gør det. Dette gøres med de fire reserverede ord i Java `try`, `throw`, `catch` og `finally`. Se Fig 5.2.1.

```
// Filnavn = input2.java

class Fejl extends Throwable
{
    public void besked()
    {
        System.out.println("Fy du er gået for langt");
    }
}


public class input2
{
    public static void main(String arg[])
        throws  java.io.IOException
        {
            int i,j;
            char stak[] = new char[10];

            try
            {
                i = 0;
                while ( (stak[i] = (char) System.in.read() ) != 13)
                {
                    i = i + 1;
                    if (i >= 10) throw new Fejl();
                }
                for ( j = i; j >= 0; j = j - 1)
                    System.out.print(stak[j]);
            }
            catch (Fejl F)
            {
                F.besked();
            }
            finally
            {
                // finally skal ikke benyttes i denne sammenhæng
            }
        }
}
C:\java input2
123456789abc
Fy du er gået for langt

C:\
```

Fig 5.2.1

På Fig 5.2.1 har vi lavet en klasse ved navn Fejl, dette er vores exception handler, eller fejlhåndteringsklasse på Dansk. For at et objekt skal kunne kastes, skal det nedarve fra klassen Throwable, det er det der sker i første linie. Funktionen af handleren skulle være lige frem, det er en ganske normal klasse med metoder.

Når vi mistænker at noget kan gå galt, kan vi sætte det ind i en try blok, og fange evt. opståede fejl med en catch blok. Selve fejlen kan vi kaste med throw.

Til sidst har vi finally blokken. Det er en blok der altid udføres, i denne blok kan man udføre oprydning som lukning af filer. Der er intet at rydde op i vores tilfælde, derfor er finally blokken tom.

I skitseform ser det hele således ud.

```
try
{
     ...
     throw  new klassenavn;
     ...
}
catch (klassenavn objektnavn)
{
     // Fejlhåndtering, fejlmeddelelser m.m.)
}
finally
{
     // Oprydning hvis nødvendig.
     // Udføres altid, uanset om der kastes eller ej.
}
```

Det klassenavn der oprettes et tilfælde af i linien

```
throw  new klassenavn;
```

skal der være en catch (klassenavn objektnavn) til. Hvis man glemmer at fange (catche) et throw, opstår en kompilerfejl. Catch blokken skal komme umiddelbart efter try blokkenes sidste slut tuborg, ellers opstår en kompilerfejl.

Der kan problemfrit kastes flere exception's inde i én try blok, (det er faktisk ganske normalt at gøre det.) men der skal være en catch for hver forskellig objekt der bliver kastet. Hvis en try blok ser således ud:

```
try
{
     throw  new klassenavn1;
     throw  new klassenavn2;
     throw  new klassenavn1;
}
```

skal der være to catch blokke, det kan se således ud:

```
catch (klassenavn1 navn1) {
     ...
}
catch (klassenavn2 navn2) {
     ...
}
```

Der må kun være en finally blok per try. Man kan udelade finally blokken, hvis man ikke ønsker at bruge den.

Ved at benytte exception's, kan man fange fejl på en standardiseret måde. Herved får man skilt fejlhåndteringskoden fra selve programkoden, hvilket gør koden nemmere at læse. Ved at benytte klasser til at håndtere fejl, får man en let genbrugbar kode.

5.3 Opgaver

Opgave 5.3.1 Skriv en metode der tager en array med `char`, og konverterer indholdet til et heltal. (`int`)

Opgave 5.3.2 Skriv et program der beder om to operander. Programmet skal derefter lægger dem sammen og udskrive resultatet på skærmen.

Opgave 5.3.3 Udvid opgave 5.3.2 således at programmet også beder om operatoren. Programmet skal kunne genkende de fire regnearter (plus, minus, gange, division)

Opgave 5.3.4 Skriv et program der kan hente en formel fra tastaturet. Formlen kan være på denne form.

32 + 12

Programmet skal kunne filtrere operator og operander fra hindanden, og udføre den opgivne regneart. Resultatet skal skrives på skærmen. Hvis der opstår en fejl under indtastningen, skal der skrives "Fejlindtastning!" på skærmen. Hvis programmet ikke kan forstå inputtet, skal der skrives "Dårlig syntaks" på skærmen.

6 Lidt teori

Hidtil har vi præsenteret nogle begreber, vist hvordan de kan benyttes, og derefter kastet læseren ud i nogle opgaver omkring emnet. Nu er det nok på tide at jonglere lidt med teorierne, og finde ud af hvad det hele egentlig betyder.

6.1 Reserverede ord

Når man erklærer variabler, metoder og klasser, skal man undgå at benytte de ord som Java benytter. Disse ord kaldes reserverede (keyword på engelsk) og må ikke benyttes. Reserverede ord kan ikke overskrives som metoder kan, forsøg på at overskrive et reserveret ord skal medføre en kompilerfejl. Tabel 6.1.1 viser en liste over reserverede ord gældende for Java version 1.1.

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Tabel 6.1.1

En hurtig optælling vil vise, at vi har arbejdet med 33 af disse ord. I resten af dette kapitel vil vi se nærmere på reglerne for brug af disse ord. Sun har annonceret en snarlig frigivelse af version 1.2. Når den kommer på gaden kan der være tilføjelser til tabellen.

Man kan opleve at en kompiler påstår at noget er deprecated, hvilket betyder at kompileren anser det for at være forældet, men brugbart. Hidtil har Java været bagudkompatibelt, men i edb branchen er det desværre ingen garanti for fremtiden. Derfor vil jeg anbefale, at unlade at bruge metoder som kompileren kalder depricated.

6.2 Kommentarer

Hidtil har vi benyttes // til at indsætte en kommentar, nogen har måske også fundet ud af, at man kan udkommetere mistænkelige linier ved at sætte // først på linien. Det har den fordel, at man kan udkommetere noget mistænkeligt, uden at skulle risikere at taste det ind igen, hvis det skulle vise sig at være rigtigt.

```
// Udkommenterer fra tegnet, og frem til først kommende linieskift
```

Der er to andre måder at indsætte kommentarer i en Java kildetekst. Den ene er ved at benytte /* til at starte en udkommenteringen, i så fald skal udkommenteringen afsluttes med */.

Den sidste måde man kan udkommentere med i Java, er ved at starte udkommenteringen med /** og slutte den med */. Dette kan af nogle kompilere benyttes til automatisk at generere dokumentation. De tre typer af udkommentering er vist på Fig 6.2.1.

```
//      Dette tegn betyder at resten af linien ikke kompileres.

/*
        Dette tegn betyder, at alt bliver udkommenteret indtil vi møder slut
        mærket
        Vi har stadig ikke mødt slut mærket
        ...
        Efter denne linie kommer slut mærket.
*/

/**
        Dette tegn betyder, at alt mellem starttegnet og sluttegnet ikke blot
        udkommenteres, men at det af nogle kompilere kan bruges til at generere
        automatisk dokumentation. Hvordan det fungerer i praksis, må du
        konferere med kompilermanualen for at finde ud af.
*/
```

Fig 6.2.1

6.3 Blokke og rækkevidde

Hidtil har vi benyttet blokmærker (tuborg) når vi skulle. Blokke kan benyttes til at specifisere rækkevidden af variabler og objekter. Prøv at se på følgende eksempel:

På Fig 6.3.1 er den 3. sidste linie udkommenteret. Hvis den ikke blev udkommenteret, ville det medføre en kompilerfejl. Dette skyldes at rækkevidden af variablen `b`, er fra `b` bliver erklæret, (linien `int b = 10;`) til det korrosponderende (samhørende) blok slut mærke. Dette er i dette tilfælde det først kommende, men det behøves ikke være tilfældet. Variablen `a` har rækkevidde fra den bliver erklæret, til det nædstsidste blok slut mærke.

Når en blok oprettes inden i en anden blok, kaldes det nestning. Det der gælder i en blok, gælder også i de blokke som blokken nester, men det modsatte er ikke tilfældet, som vi så i Fig 6.3.1.

```
// Filnavn = blok1.java
// Programmet demonstrerer rækkevidde begrebet
// Prøv at indkommentere kommentaren og
// kompiler koden. (Det skal give en fejlmelding)

public class blok1
{
        public static void main(String arg[])
        {
                int a = 3;

                System.out.println(a);
                {
                        int b = 4;

                        System.out.println(a + ":" + b);
                }
//                      System.out.println(a + ":" + b);
        }
}
```

```
C:\java blok1
3
3:4
C:\
```

Fig 6.3.1

Dette er årsagen til at vi prøver at have de korrosponderende blok mærker til at stå med samme indrykning, det gør ganske enkelt rækkevidden tydeligere.

Blokke benyttes også til at samle det der skal udføres, prøv at sammenligne eksemplerne i Fig 6.3.2.

Eksempel a kan være lidt forvirrende, det skyldes udskrivningslinien har en indrykning der antyder, at den skal udføres de 10 gange vi kører rundt i `for` løkken, og det er ikke tilfældet. Hvis vi vil have udskrevet “Slut på det” for hver gennemgang af løkken, skal det udføres som i eksempel b. Vi får altså mere udført i `for` løkken, hvis vi lader løkken udføre en blok, i stedet for en linie.

```
// Eksempel a
// Dette er kun en del af et program,
// Kan ikke kompiles særskilt

int i;

for (i = 1; i <= 10; i = i + 1)
      System.out.println(i);
                System.out.println(“Slut på det.”);

// Eksempel b
// Dette er kun en del af et program,
// Kan ikke kompiles særskilt

int i;

for (i = 1; i <= 10; i = i + 1)
{
      System.out.println(i);
                System.out.println(“Slut på det.”);
}
```

Fig 6.3.2

Vi kan udvidde problematikken ved at forlange at to objekter skal kunne “snakke sammen”. (Med et fint ord kaldes det en assosiation)

For at skabe en kommunikation mellem to objekter, er det nødvendigt at være sig bevidst om rækkevidden. Lad os forestille os vi har et objekt ved navn Hansen, og et objekt ved navn Jensen, begge er tilfælde af klassen Person.

Man kunne fristes til at lave noget i stil med Fig 6.3.2

```
// Psedokode: Kan ikke kompiles

objekt Hansen
{
     Jensen = new Person();
}
```

Fig 6.3.2

Så har man jo skabt et objekt Hansen og et objekt Jensen. Problemet er bare, at Hansen kan se Jensen, og alt hvad der ikke er protected og private derinde, men Jensen aner intet om Hansens eksistens. Hansen har rækkevidde ind i Jensen, det modsatte er ikke tilfældet.

Vi er i stedet nød til at have et rum hvor kommunikationen kan bevæge sig. En måde at skabe dette rum på er vist på Fig 6.3.3.

```
// Filnavn = sniksnak.java
// Demonstrerer kommunikation mellem objekter

class Person
{
    private int tmp;
    private String Pnavn;

    Person(String navn)
    {
        Pnavn = new String(navn);
        tmp = 0;
    }

    public void saet_tmp(int a)
    {
        tmp = a;
    }

    public int hent_tmp()
    {
        return tmp;
    }

    public void skriv_tmp()
    {
        System.out.println("Hos " + Pnavn + " er tmp = " + tmp);
    }
}


public class sniksnak
{
    public static void main(String arg[])
    {
        Person Hansen = new Person("Hansen");
        Person Jensen = new Person("Jensen");

        Hansen.saet_tmp(20);
        Hansen.skriv_tmp();
        Jensen.skriv_tmp();
        Jensen.saet_tmp(Hansen.hent_tmp());
        Hansen.skriv_tmp();
        Jensen.skriv_tmp();
      }
}
```

C:\java sniksnak
Hos Hansen er tmp = 20
Hos Jensen er tmp = 0
Hos Hansen er tmp = 20
Hos Jensen er tmp = 20

C:\

Fig 6.3.3

I programmet sniksnak oprettes de to objekter der skal kunne snakke sammen i et fælles rum. Dette rum er i dette tilfælde main, men det kan ske i et hvilken som helst metode.

Objekterne Hansen og Jensen kan stadig ikke "se" hindanen, men rummet main kan se begge. Det fælles rum har ikke adgang til de private variabler `Pnavn` og `tmp`, de er nemlig korrekt indkapslet i klasserne. I stedet kommer vi til variablerne på en veldefineret måde, gennem klassens metoder.

I eksemplet ønsker Jensen at kende værdien i Hansen's variabel tmp. Man kan fristes til at gøre `tmp` public og så bare skrive noget i retning af:

```
tmp = Hansen.tmp;   // Ganske uheldigt !!!
```

Det kan ikke lade sig gøre, fordi Jensen ikke kan "se" Hansen. I øvrigt ville det være at invitere til spaghettiprogrammering. Der er intet behov for at `tmp` skal være public.

I stedet lader vi Jensen bede pænt om data. Til det formål benytter vi metoden `saet_tmp(int a)` i klassen person. For at få informationen ud af Hansen, benytter vi metoden `int hent_tmp()`, der jo netop er sat i verden med det ene formål, at svare på spørgsmålet, hvad er tmp?

Det pænt formulerede spørgsmål, "vil du fortælle mig hvad værdi tmp har os dig?", ser således ud i Java:

```
Jensen.saet_tmp(Hansen.hent_tmp());
```

Vi sikrer os altså at Jensen spørger Hansen gennem sin metode `saet_tmp(int)`, Hansen svarer pænt via sin metode `int hent_tmp()`. Det hele foregår i rummet sniksnak.

6.4 Magiske tal

Vi har tidligere været inde på at magiske tal kan skabe problemer i et program. Årsagen til at man kalder dem magiske er jo netop, at de kan få de mest magiske ting til at ske i et program. Prøv at forestille dig at du har et stort program, med mange `for` løkker, alle løkkerne skal starte det samme sted. Det kræver ikke den store fantasi at forestille sig hvad der vil ske hvis man ønsker at rette startværdien. Man kan ikke finde alle steder, derfor glemmer man at rette i et eller flere steder, med programfejl til følge der måske først opdages ude ved brugeren.

Det gælder altså om at undgå magiske tal, der jo dybest set er konstanter. I Java kan vi erklære en variabel som værende konstant, ved at benytte det reserverede ord `final`. Når en variabel er erklæret som `final`, så må den ikke ændres. (Hvorfor det strengt taget ikke længere er en variabel) Derfor skal en konstant tilskrives når den erklæres, der er ingen mulighed for senere at tilskrive den. Hvis man forsøger at tilskrive en konstant efter den er erklæret, vil kompileren opfatte det som en fejl, og afgive en fejlmeddelelse under kompileringen.

```
// Filnavn = konst1.java

import java.lang.Math;

class do_me
{
    public void print(double data[], int max)
    {
        int i;

        for (i = 0; i < max; i++)
            System.out.println(data[i]);
    }
}

public class konst1
{
    public static void main(String arg[])
    {
        int i;
        final int MAX = 5;
        double tal[] = new double[MAX];
        do_me do_meWell = new do_me();

        for (i = 0; i < MAX; i++)
            tal[i] = Math.random();
        do_meWell.print(tal,MAX);
    }
}
```

```
C:\java konst1
0.xxxxxxxxxxxxxx
0.xxxxxxxxxxxxxx
0.xxxxxxxxxxxxxx
0.xxxxxxxxxxxxxx
C:\
```

xxxxxxxxxxxxx = tilfældige tal.

Fig 6.4.1

Programmet konst1.java på Fig 6.4.1 demonstrerer brugen af konstanter. Bemærk at konstanter i Java skrives med store bogstaver. Dette er en tradition der er kendt fra andre sprog, men kompileren er faktisk ligeglad. Det har dog den fordel, at man straks kan se noget er en konstant. At man samtidig kan have en variabel med samme navn som en konstant, men skrevet med små bogstaver, kan i nogle tilfælde være praktisk. (Men det kan også være forvirrende)

Programmet viser det praktiske i at have konstanten MAX. Den bruges til at erklære størrelsen af et array, derefter bruges den som grænse i `for` løkken, og minsanten om ikke også den overføres til print metoden inde i `do_meWell`.

Inde i printmetoden er `max` ikke en konstant, det vil altså være muligt at ændre på indholdet af `max` inde i printmetoden, uden at påvirke `MAX` i main metoden. Man kan diskuttere dette valg, da det nok ikke er nogen god ide at ændre på `max` inde i printmetoden.

Hvis man vil have at `max` også skal være en konstant inde i printmetoden, skal den erklæres som vist på Fig 6.4.2

```
class do_me
{
    public void print(double data[], final int MAX)
    {
        int i;

        for (i = 0; i < MAX; i++)
            System.out.println(data[i]);
    }
}
```

Fig 6.4.2

Bemærk i øvrigt, at konstanterne har navne der leder tankerne hen mod hvad de bruges til, det er også med til at øge læserværdien af programmet.

Opgave 6.4.1 Hvad vil der ske i med MAX i main (Fig 6.4.1) hvis max ændres i printmetoden? Prøv det !

6.5 Mere om for

For løkken kan mere end vi hidtil har set, Fig 6.5.1 viser et program der tæller to variabler mod hindanden i en for løkke.

```
// Filnavn = nonmagi.java


public class nonmagi
{
        public static void main(String arg[])
        {
                final int NEDRE =  1;
                final int OEVRE = 10;
                int i,j;

                for (i = NEDRE,j = OEVRE; i <= j; i++,j--)
                        ;
                System.out.println("Variablerne i og j møder hinanden ved ");
                System.out.println("i = " + i + " og j = " + j);
        }
}
```

C:\java for1
Variablerne i og j møder hindanen ved
i = 6 og j = 6

C:\

Fig 6.5.1

Her tilskrives variablerne `i` og `j` begge i et huk, det er noget anderledes fra hvad vi har set før.

```
for (i = 1,j = 10; i <= j; i++,j--)     // 2 variabler ??
                          ;             // Dette er også nyt
```

Vi kan altså arbejde med flere variabler i `for` løkken, de skal blot sepereres med et komma. Vi behøver ikke at benytte det samme antal i initialisreingsdelen og evalueringsdelen af løkken, men vi kan gøre det.

Den anden linie, (linien med kun et semikolon) betyder blot, at vi ikke skal lave noget inde i løkken, det er jo ikke nødvendigt. Årsagen til at semikolon er flyttet ned på en linie for sig selv er, at det derved bliver tydeligt at programmøren ikke ønsker at foretage sig noget i løkken. Man kunne også have skrevet:

```
for (i = 1,j = 10; i <= j; i++,j--);
```

Kompileren kan ikke se forskel, men vi mennesker kan have svært ved at få øje op på semikolon, og derfor tro der står:

```
for (i = NEDRE,j = OEVRE; i <= j; i++,j--)
```

```
            System.out.println("...");
```

Dette kan igen få os til at fare ned og lave indrykning på næste linie, for at gøre koden mere læseværdig. Derefter kan vi så bruge en time eller to på at finde ud af hvorfor programmet ikke virker som vi tror det gør. Derfor foreslår jeg at man sætter et semikolon til en tom løkke på en linie for sig selv. (Dette er i modstrid med kodestandarden, den foreslår at semikolon altid sættes på samme linie som udtrykket)

En anden facilitet i Java er, at variabler kan erklæres hvor man bruger dem. Der foregår en lidenskabelig diskution i programmørkredse hvorvidt det nu er så smart. Men specielt i følgende eksempel mener jeg det kan være ganske praktisk. Se Fig 6.5.2.

```
// Filnavn = middevaerdi.java

import java.lang.Math;

class min_Math
{
    public double middel(double data[], int max)
    {
        double tmp = 0;
        for (int i = 0; i < max; i++)
            tmp = tmp + data[i];
        return tmp/max;
    }
}

public class middelvaerdi
{
    public static void main(String arg[])
    {
        int i;
        final int MAX = 5;
        double tal[] = new double[MAX];
        min_Math math = new min_Math();

        for (i = 0; i < MAX; i++)
            tal[i] = Math.random();
         System.out.print(math.middel(tal,MAX));
    }
}
```

C:\java middelvaerdi
0.xxxxxxxxxxxx
C:\

Fig 6.5.2

Programmet middelvaerdi.java foretager sig ikke ret meget andet, end at generere et sæt tilfældige tal, for derefter at beregne middelværdien af disse tal.

I Metoden `middel` sker der noget nyt, linien:

```
for (int i = 0; i < max; i++)
```

medfører at variablen `i` bliver oprettet i `for` løkkens initialiseringsdel. I Dette tilfælde har variablen rækkevidde i den løkke hvor den er erklæret. Hvis middelmetoden udviddes med linien

```
tmp = i; // FEJL !!!
```

som vist på Fig 6.5.3 vil der opstå en fejl under kompileringen. Dette skyldes at `tmp` forsøges tilskrevet med `i`, der er uden for rækkevidde.

```
public double middel(double data[], int max)
{
      double tmp = 0;

      for (int i = 0; i < max; i++)
           tmp = tmp + data[i];
      tmp = i; // FEJL !!!
      return tmp/max;
}
```

Fig 6.5.3

Opgave 6.5.1 Hvad vil der ske i programmet nonmagi,(Fig 6.5.1) hvis variablerne `i` og `j` blev tilskrevet prefix i stedet for postfix?

Opgave 6.5.2 I programmet middevaerdi bør middelværdien nærme sig 0,5 des større `MAX` bliver. Undersøg hvor stor `MAX` skal være, for at det er tilfældet.

Opgave 6.5.3 Tilpas programmet nonmagi.java, således at grænseværdierne skal indtastes fra tastaturet. (Husk, de må i så fald ikke erklæres som konstanter)

6.6 Mere om if

Med `if` kan vi tage beslutninger. If kan udviddes med blokke hvis vi vil have flere ting udført hvis en given betingelse er opfyldt. Det udføres som vist på Fig 6.6.1. Bemærk at vi kan udelade elsedelen helt, hvis vi ikke har noget at bruge den til.

```
if (Dette er sandt)
{
     Gør dette
     Gør også dette
     Dette skal også gøres
}    // Her ophører if, og en evt. else kan starte.
else // Kan benyttes, skal ikke
{
     Dette udføres hvis if betingelsen ikke er sand
     ..
     Eksekveringen ophører når vi møder blokslut
}
     // Her er vi helt ude af if ... else strukturen.
```

Fig 6.6.1

Ofte kan det være nødvendig at have mere end én betingelse for at udføre noget, det kaldes at neste. (to nest = at bygge rede) En nestet if opbygning er vist på Fig 6.6.2.

```
if (a == b)
     if (b == c)
     {
          Gør dette
          Gør dette
     }
```

Fig 6.6.2

Vi kan i stedet benytte AND operatoren, således at det hele udføre ud fra en sammensat betingelse. (Se Fig 6.6.3)

```
if (a == b && b == c)
     {
          Gør dette
          Gør dette
     }
```

Fig 6.6.3

De to og tegn udgør tilsammen and operatoren. If sætningen udføres i dette tilfælde kun hvis a er lig med b, og hvis b = c. (Heraf er det indlysende at a er lig med c, men det er eksemplet uvedkommende.)

Hvis vi i stedet vil udføre noget hvis blot én ud af to (eller flere) betingelser er sand, kan det udføres som på Fig 6.6.4.

```
if (a == b || b == c)
     {
          Gør dette
          Gør også dette
     }
```

Fig 6.6.4

De to lodrette streger er or operatoren. (or betyder eller) Det der er inde i if stætningen udføres altså, hvis a er lig med b, eller hvis b er lig med c.

Det kunne naturligvis også være lavet som vist på Fig 6.6.5

```
if (a == b)
     {
          Gør dette
     }
if (b == c)
     {
          Gør dette
     }
```

Fig 6.6.5

Ved at bruge and og or operatorene kan man undgå at bruge nestede if sætninger. Fig 6.6.6 viser et eksempel hvor det kan give andledning til forvirring, hvis man nester.

```
if (a == b)
     if (b == c)
     {
          Gør dette
          Gør dette
     }
else
{
     Så skal dette gøres
}
```

Fig 6.6.6

Dette vil være let at misforstå, for hvornår udføres else blokken ? I eksemplet ovenfor udføres else blokken hvis b er forskellig fra c, men det kan være svært at se ud fra indrykningen. Den slags programmering kan give andledning til fejl det kan tage meget lang tid at finde.

Med lidt frihed kan man vel oversætte nestning til "en værre redelighed". Men man kan ikke

programmere uden at neste. Så mit råd til nye programmører er som altid KIS. (Keep It Simple) Det kan bedst forklares med følgende.

Enkle systemer laver enkle fejl, komplicerede systemer laver komplicerede fejl.

Opgave 6.6.1 Skriv et program der beder om et tal fra tastaturet. Hvis det indtastede tal befinder sig i intervallet fra og med fire, til og med syv, skal der udskrives TRÆFFER, ellers skal der udskrives FORBI.

Opgave 6.6.2 Skriv koden til det klassiske spil høj/lav. Programmet skal "tænke" på et tal mellem nul og hundrede, derefter skal det bede brugeren om at gætte tallet. Brugeren skal have lov til at gætte så mange gange det er nødvendigt for at finde tallet. For hvert gæt skal programmet oplyse brugeren, om gættet er for højt, for lavt eller rigtigt. Programmet skal stoppe når brugeren har fundet tallet.

6.7 Switch

Hvis man ønsker at gøre én ting ud af mange mulige, kan man bygge sit program op af mange if sætninger. Et program der skal genkende en person ud fra en kode kan se således ud.

```
if (ind == 1)
     System.out.println("Hej Hans");
if (ind == 2)
     System.out.println("Hej Jens");
if (ind == 3)
     System.out.println("Hej Mette");
```

Når der er behov for at tage én beslutning ud af mange, kan brugen af if sætningen hurtig blive uoverskuelig, derfor er der udviklet en switch. Med det samme eksempel som før, ville en switch se ud som vist i Fig 6.7.1.

```
// Filnavn = switch1.java

public class switch1
{
     public static void main(String arg[])
     {
          int ind = 2;

          switch (ind)
          {
               case 1 : System.out.println("Hej Hans");
                        break;
               case 2 : System.out.println("Hej Jens");
                        break;
               case 3 : System.out.println("Hej Mette");
                        break;

               default : System.out.println("Dig kender jeg ikke");
          }
     }
}
C:\java switch1
Hej Jens

C:\
```

Fig 6.7.1

En switch fungerer ved at kigge på en variabel, i dette tilfælde `ind`, for at tage en beslutning ud fra hvad variablen indeholder. Dette sker i linien:

```
switch (ind)
```

Denne linie skal altid følges af en blok start, switchen stopper ved udgangen af det tilhørende blokslut. Inde i blokken kan der tages en beslutning, alt efter hvad der er sandt. (Hvis noget) For hver `case` (sag) kan der udføres et eller andet, hvis der skal udføres mere end én ting, skal der benyttes blokke. Hver `case` kan, men skal ikke afsluttes med en `break`.

Når programmet møder en `break` inde i en `switch`, vil eksekveringen i switchen stoppe og fortsætte umiddelbart efter switchen. Hvis en `case` ikke afsluttes med en `break`, vil switchen fortsætte, dette kan ofte føre til en ustabil situation, derfor må det frarådes.

I Fig 6.7.1 betyder linien:

```
case 1 : System.out.println("Hej Hans");
         break;
```

Hvis variablen `ind` er lig med værdien `1`, så skal der skrives “Hej Hans” på skærmen, derefter møder vi en `break`, hvilket betyder at switchen skal forlades betingelsesløst. Det vil altså ikke blive undersøgt om `ind` også er lig med 2.

Hvis ingen betingelse i switchen er opfyldt, udføres default: linien. Denne linie kan udelades. Skal der udføres mere end én ting inde i default, skal hele default udtrykket ind i en blok.

I Fig 6.7.1 betyder linien:

```
default : System.out.println("Dig kender jeg ikke");
```

At hvis programmet er “faldet igennem” uden at nogen betingelse er opfyldt, vil der blive skrevet

```
Dig kender jeg ikke
```

På skærmen.

Opgave 6.7.1 Skriv et program der ud fra et månedsnummer udskriver månedens navn. Januar er måned 1.

Opgave 6.7.2 Tilpas programmet fra opgave 5.2.3, således at operatoren bestemmes ved hjælp af en switch. Programmet skal opdage en ugyldig operator.

Opgave 6.7.3 Udvid opgave 3.8.1 med et passende antal metoder, således at data til de enkelte objekter skal indtastes fra tastaturet, i stedet for at overføres til konstruktøren. Valg af indtastning skal foregå fra en menu, hvor brugeren skal vælge hvad han ønsker at indtaste.

6.8 Mere om metoder

En metode er faktisk bare en en blok med et navn, og mulighed for at overføre parametre til blokken. Vi kan sammenligne en metode med “black box” (sort kasse) begrebet. En black box gør et stykke arbejde for os, uden at vi behøver at overveje hvordan den gør det. Hvis vi drejer tændingsnøglen i en bil, så sker der et ud af to. Enten starter motoren og vi kan køre, eller også starter motoren ikke, og så må vi rette fejlen. I denne sammenhæng kan vi se bilens startsystem som en black box. I pseodokode kan det se således ud:

```
if ( !start_bil() )
     ret_fejl();
koer();
```

Man behøver ikke at være et geni for at finde ud af hvad koden går ud på, og hvordan den arbejder. Hvordan en evt. fejl rettes er for nuværende komplet ligegyldig, det har vi folk (metoder) til at tage sig af.

En metode skal erklæres på dene måde:

```
Metodeerklæring
{
     Metodekrop
}
```

En metode indeholder altså en metodeerklæring, efterfulgt af en metodekrop, indeholdt i en blok.

8.8.1 Metodeerklæring

En metodeerklæring skal som minimum indeholde metodens navn, og dens returtype.

```
returtype metodenavn()
{
     // return returtype
}
```

Returtypen er en af typerne beskrevet i tabel 2.2.1 en brugerdefineret type (objekt), eller `void` (void = ingenting) hvis ikke metoden skal aflevere data. Hvis `void` vælges som returtype, er det en fejl at bruge `return` udtrykket i metodkroppen. Hvis metoden har en returtype, er det en fejl ikke at benytte r`eturn` udtrykket i metode kroppen.

Returtyper kan tilpasses med en cast.

Metoder skal altid returnere samme type som er defineret i metodeerklæringen, alt andet er en fejl. Følgende vil medføre en kompilerfejl

```
int mintst()
{
     float pi = 3.1459;

  return pi;   // Fejl, illegal returtype
}
```

Hvis returtypen er en klasse kan metoden returnere objekter af samme type, og af alle nedarvede typer. Da alle klasser nedarver fra Object klassen, kan man bruge Object som returtype. Det vil medføre at man kan returnere et hvilket som helst objekt, da de jo alle på en eller anden måde arver fra Object klassen. Programmet i_live.java viser denne teknik.

```
// Filnavn = i_live.java

class tst
{
    Object minMetode()
    {
        String tmp = new String("Land i sige");

        return tmp;
    }
}

public class i_live
{
    public static void main(String arg[])
    {
          tst a = new tst();
          String z;

          z = (String) a.minMetode();
          System.out.println(z);
    }
}
```

```
C:\java i_live
Land i sigte

C:\
```

Fig 8.8.1.1

Programmet i_live.java viser også en anden sjov egenskab i Java. Objektet `tmp` der bliver skabt i klassen `a`, eksisterer selv om det egentligt ser ud som om det er forsvundet uden for rækkevidde. Årsagen til at objektet kan tilgåes i main er, at i Java slettes objekter af en garbage collector (gc). Garbage colectoren er så smart, at den ikke sletter objekter der er refereret til, hvorfor objektet fortsætter med at leve, selv om det objekt der skabte den forsvinder ud af rækkevidde.

Udtrykket `return x;` kan bruges flere gange i en metode, dette er vist i Fig 6.8.1.2

```
// Filnavn = rtntst.java

public class rtntst
{
        public static int min(int a)
        {
                if (a == 0)
                        return 1;
                else
                        return 0;
        }
        public static void main(String arg[])
        {
                System.out.println(min(0));
        }
}
```

```
C:\java tst
1

C:\
```

Fig 6.8.1.2

Metodenavnet er enhver kombination af bogstaver, tal og gyldige tegn. Metodenavne må dog ikke begynde med et tal.

En fuldstændig metodeerklæring ser således ud:

```
[accessSpecifier] [static] [abstract] [final] [native]
[synchronized]
     returType metodeNavn ([parameterliste])
     [throws exceptionListe]
```

Det der er angivet i firkantede klammer [ ] kan udelades. Flere af udtrykkende er gennemgået, resten vil blive gennemgået senere.

6.8.2 Metodekroppen

I metodekroppen er der erklæringer og udtryk. Erklæringer kan se således ud:

```
int a,b,c;
int d,e,f;
char ch;
String minStr;  // Erklæret men ikke defineret.
```

Fig 6.8.2.1

Med en erklæring "fødes" en variable eller en referense til et objekt. En erklæret variabel eksisterer fra den bliver født (erklæret) til enden af den blok den blev erklæret i. Fig 6.8.2.2 viser et eksempel på legal og illegal brug af variabler.

```
int a,b;              // Erklæring.
String minStr;        // Erklæret men ikke defineret.

a = 3; b = 2 * a;     // Lovlig initialisering.
c = a + b;            // FEJL: c er ikke erklæret, og kan derfor ikke benyttes.
int c;                // Erklæring er lovlig, men ubrugelig i dette eksempel.
```

Fig 6.8.2.2

Variabler kan initialiseres i erklæringsdelen som tidligere vist, men det er nyt at vi ikke definerer et tilfælde af minStr. Ind til nu har vi næsten kun brugt teknikken:

```
String minStr = new String();
```

Fig 6.8.2.3

men dette kan deles op i flere linier, fjernt fra hinanden, det kan se således ud:

```
String minStr;           // Erklæring.
minStr = new String();   // Definering.
```

Fig 6.8.2.4

Variabler og objekter kan erklæres overalt i en metode, men det er kutyme at erklære dem i metodekroppens første del. Der er dog ved at brede sig en idé i programmørkredse om, at det kan være en fordel at erklære en variabel hvor den bliver brugt. Argumentet herfor er, at så kan man se variabletypen når man skal burge den. Et eksempel på en sådan teknik er givet i Fig 6.8.2.5

```
// Filnavn = for3.java

public class for3
{
        public static void main(String arg[])
        {
                final int MAX = 5;
                for (int i = 0; i <= MAX; i++)
                        System.out.println(i);
        }
}
```

```
C:\java for3
0
1
2
3
4
5

C:\
```

Fig 6.8.2.5

I Fig 6.8.2.5 er variablen `i` erklæret og initialiseret inde i `for` løkken. Dette medfører at variablen `i` kan bruges inde i `for` løkken, men ikke udenfor. Dette uanset at `for` løkken ikke har nogen blok, men blot udfører et udtryk. Man siger at `i`'s rækkevidde er løkken.

En variabel må ikke erklæres to gange i en metode. Fig 6.8.2.6 viser en illegal variant af Fig 6.8.2.5:

```
// Programmet kan ikke kompiles

public class for3
{
   public static void main(String arg[])
   {
     int i;

     for (int i = 0; i <= 10; i++)      // FEJL !!!
     System.out.println(i);
   }
}
```

Fig 6.8.2.6

Fejlen i Fig 6.8.2.6 opstår fordi variablen `i` forsøges erklæret to gange. Kompileren vil fejlmelde ved `for` løkken, men fejlen kan rettes ved at slette den første erklæring af variablen `i`. Dette er et tilfælde på, at fejlen ikke nødvendigvis er på den linie hvor kompileren melder den. (Retfærdigvis skal det nævnes, at det er en smagssag hvilken af de to erklæringer af `i` der skaber fejlen)

this

Hvis man finder det nødvendig at have to variabler med samme navn i en klasse, den ene gældende for klassen, den anden for metoden, kan der opstå misforståelser ved brug. Dette undgåes ved at bruge `this.` (this punktum) når man vil referere til klassens variabel. Fig 6.8.2.7 viser et eksempel på brug af `this.`

```
class tst_klasse
{
        int a;   // Erklæring

        tst_klasse()  // Konstruktør
        {
                a = 5;
        }

        void var_tst()
        {
                int a = 1; // Nok en erklæring af a

                System.out.println(a);
                System.out.println(this.a);
        }
}

public class this1
{
        public static void main(String arg[])
        {
                tst_klasse min_tst = new tst_klasse();
                min_tst.var_tst();
```

```
        }
}

C:\java this1
1
5

C:\
```

Fig 6.8.2.7

På Fig 6.8.2.7 er variablen `a` erklæret to gange i klassen `tst_klasse`. Den første erklæring af `a` har rækkevidde fra den bliver eklæret, til afslutning af den blok den er erklæret i, i dette tilfælde er det hele klassen. Metoden `var_tst` erklærer nok en variabel ved navn `a`, dennes rækkevidde er hele metoden, den eksisterer ikke uden for metoden `var_tst`. For at metoden kan skelne mellem variablerne, må den bruge `this`. For metoden `var_tst` gælder følgende:

```
System.out.println(a);   // a = 1
System.out.println(this.a);   // a = 5
```

Ved at bruge `this` kan vi sikre os, at vi har fat i klassen's variabel `a`. Det er ikke en fejl at bruge t`his` til at refere til klassens variabler, selv om variablen ikke findes i den lokale metode. Nogen mener endda at det gør programmet mere læseligt hvis man altid benytter `this`.

Det er også muligt at bruge `this.` hvis man ønsker at anvende en metode fra samme klasse. Dette er der vist et eksempel på i Fig 6.8.2.8

```
class tst_klasse
{
        int a;

        void tst()
        {
                a++;
        }

        void var_tst()
        {
                a = 1;

                System.out.println(a);
                this.tst();
                System.out.println(a);
        }
}

public class this2
{
        public static void main(String arg[])
```

```
        {
                tst_klasse min_tst = new tst_klasse();
                min_tst.var_tst();
        }
}
```

```
C:\java this2
1
2

C:\
```

Fig 6.8.2.8

Af Fig 6.8.2.8 fremgår det, at fra metoden `var_tst` kaldes metoden `tst`. Brugen af `this`. kan være specielt praktisk når man arbejde med overskrevne metoder og variabler.

6.8.3 Overførsel af data til metoder

Hidtil har vi blot overført data til en metode, og benyttet returværdien. Dette er tilstrækkeligt til mange opgaver, men er også en begrænsning. Vi vil nu se på en teknik hvor vi kan "returnere" flere parametre end blot returtypen. Kort fortalt går det ud på, at alle argumenter der er et objekt ikke overføres til metoden, men bliver brugt direkte af metoden. Dette kaldes også "call by reference". Navnet skyldes at det ikke er selve objektet der overføres til metoden, men en reference til objektet. Dette medfører, at metoden arbejder direkte på objektet.

```
// Filnavn = met_obj.java


class demo_klasse
{
        int Heltal;
}

class tst_klasse
{
        void tst(int a, demo_klasse Demo)
        {
                System.out.println("a      = " + a);
                System.out.println("Heltal = " + Demo.Heltal);
                a++;
                Demo.Heltal++;
                System.out.println("a er nu           = " + a);
                System.out.println("Demo.Heltal er nu = " + Demo.Heltal);

        }
}

public class met_obj
{

        public static void main(String arg[])
        {
```

```
                int i = 10;
                tst_klasse  mintst  = new tst_klasse();
                demo_klasse minDemo = new demo_klasse();

                System.out.println("Program start");
                minDemo.Heltal = 33;
                mintst.tst(i,minDemo);
                System.out.println("Efter tst_klasse");
                System.out.println("i              = " + i);
                System.out.println("minDemo.Heltal = " + minDemo.Heltal);

        }
}
```

```
C:\java met_obj
Program start
a      = 10
Heltal = 33
a er nu           = 11
Demo.Heltal er nu = 34
Efter tst_klasse
i              = 10
minDemo.Heltal = 34

C:\
```

Fig 6.8.3.1

I programmeringssprog som C/C++ svarer det til at vi overfører adressen på objektet, og ikke selve objektet. (Java gør faktisk det samme, men programmøren har ikke adgang til at arbejde direkte med adresser)

Fig 6.8.3.1 viser at variablen `i` (fra main) ikke ændrer værdi i main, selv om dens parameter i metoden `tst` opdateres. Derimod ændrer `Heltal` værdi i main, dette skyldes at `Heltal` er en del af det overførte objekt.

Det er værd at bemærke, at der oprettes ingen objekter i `tst` metoden, men der arbejdes på et objekt kaldet `Demo`. `Demo` i `tst` er en reference til objektet `minDemo` i `met_obj` klassens main metode.

Call by reference kan også benyttes hvis vi overfører et array. Dette er der vist et eksempel på i Fig 6.8.3.2.

```
// Filnavn = tillaeg.java

class tst_klasse
{
        void tilSkriv(double lokal[], final int TOP)
        {
                for (int i = 0; i < TOP; i++)
                        lokal[i] = lokal[i] * 10.0;
        }
```

```
}

public class tillaeg
{
        public static void main(String arg[])
        {
                final int TOP  = 10;
                double mitArray[] = new double[TOP];
                tst_klasse a = new tst_klasse();

                for (int i = 0; i < TOP; i++)
                        mitArray[i] = Math.random();

                for (int i = 0; i < TOP; i++)
                        System.out.println(mitArray[i]);
                a.tilSkriv(mitArray,TOP);
                System.out.println("Så ganger vi med 10");

                for (int i = 0; i < TOP; i++)
                        System.out.println(mitArray[i]);
        }
}
C:\java tillaeg
0.7260009682211221
0.22374124600995393
0.023308863314096318
0.3287538950961747
0.5270317516524354
0.473648679432339
0.8218146572194188
0.21989187742353133
0.01202867579305189
0.7172930685441007
Så ganger vi med 10
7.260009682211221
2.2374124600995393
0.23308863314096318
3.287538950961747
5.270317516524354
4.73648679432339
8.218146572194188
2.198918774235313
0.1202867579305189
7.172930685441007

C:\
```

Fig 6.8.3.2

Metoden `tilskriv()` laver intet andet end at gange indholdet af hvert array element med ti. årsagen til at vi skriver `10.0` og ikke bare `10` er, at `10.0` er et kommatal, tallet `10` opfatter kompileren som et heltal (`int`), så kompileren vil forlange en cast fra `int` til `double`, hvis vi ikke skriver `10.0`.

Som det ses er den ændring der udføres i den kaldte metode også gældende i main.

Opgave 6.8.3.1 Tilpas programmet tillaeg på Fig 6.8.3.2 således, at den faktor der skal ganges på overføres som argument.

Opgave 6.8.3.2 Skriv en metode `int tael_bogstaver(String str)`. Metoden skal tælle antallet af bogstaver (store som små) i `str` og returnere antallet til den kaldende metode. Tal, mellemrum og specialtegn som punktum og komma må ikke tælles med. Test metoden.

Opgave 6.8.3.3 Skriv en metode `int tael_mellemrum(String str).` Metoden skal kun tælle antallet af mellemrum i `str`, og returnere det antal til den kaldende metode. Test metoden.

Opgave 6.8.3.4 Fremstil én metode der erstatter begge metoderne `tael_bogstaver(String str)` og `tael_mellemrum(String str)`. Test metoden.

6.9 Mere om klasser

En klasse benyttes til at samle variabler, metoder og andre klasser på en systematisk måde. En klasse erklæres på følgende måde

```
KlasseErklæring
     {
          Klassekrop
     }
```

En minimumsklasseerklæring består af det reserverede ord `class`, efterfulgt af klassens navn. I klassekroppen kan der erklæres variabler, metoder og fra Java version 1.1 kan der også erklæres klasser inde i en klasse.

En fuldstændig klasseerklæring kan se således ud:

```
[ modifier ] class Klassenavn [ extends Superklassenavn] [ implements
InterfaceNavne ]
{
                    . . .
}
```

Det der står i firkantede paranteser kan udelades. Hvis klassen ikke defineres til at arve noget, vil den per automatik arve fra den overordnede klasse Object. Alle klasser arver fra Object, om ikke direkte, så via andre klasser.

6.9.1 Klasse modifiers

En klasse kan være erklæret som public, abstract eller final. I hver fil må der ikke være mere end én klasse der er erklæret som public. Navnet på denne klasse, er det samme som filnavnet med tilføjelsen ".java". Hvis der ønskes samling af flere klasser med en naturlig sammenhæng, kan det gøres ved at bruge pakker.

En abstract klasse er en klasse der ikke må kunne skabes et tilfælde (objekt) af.

En klasse der er erklæret som `final`, kan ikke nedarves af andre klasser. Det betyder at hvis en programmør ønsker at gøre brug af en klasse der er erklæret som `final`, så er programmøren nød til at oprette et tilfælde (objekt) af den klasse.

Hvis en metode erklæres som `final`, betyder det at metoden ikke kan overskrives af nedarvede metoder.

Fig 6.9.1.1 viser et eksempel på brug af `final`. Bemærk, koden kan ikke kompileres.

```
// Filnavn = final1.java
// KAN IKKE KOMPILES
// Eksempel til belysning af final

class Min_Klasse
{
        final void metode1() {   }

        void metode2()  {   }
}

final class Barnet extends Min_Klasse
{
        int  metode1()  {   }                        // Illegal metode1 er final
        void metode2()  {   }
}

final class Mit_Barn extends Barnet     // Illegal Barnet er final
{
        void metode1()  {   }                        // Illegal metode1 er final
        void metode2()  {   }
}

public class final1
{
        public static void main(String arg[])
        {
                Min_Klasse a = new Min_Klasse();
                Barnet     b = new Barnet();    // Helt legalt
        }
}
```

Fig 6.9.1.1

6.9.3 objekt og klasse medlemmer

Der findes to slags klassemedlemmer, dem der har rækkevidde i objektet, og dem der har rækkevidder i klassen. Et eksempel på en variabel med rækkevidde i objektet er vist på Fig 6.9.3.1

```
class Min
{
     int a;
     .....
}
```

Fig 6.9.3.1

Variablen `a` eksisterer ikke før der oprettes et tilfælde af klassen, og forsvinder når objektet destrueres. Variablen `a` eksisterer altså kun så længe der findes et tilfælde af klassen. Hvert tilfælde af klassen (hvert objekt) vil have sin egen udgave af `a`.

Hvis vi i stedet erklærer variablen `a` som værende `static`, som vist på Fig 6.9.3.2, så vil der kun blive oprettet ét tilfælde af variablen `a`, uanset hvor mange objekter der oprettes af klassen. Dette tilfælde vil dække alle tilfælde (objekter) der bliver skabt af klassen. Man siger at en statisk variabel

er et klasse medlem, fordi den gælder for hele klassen. Et ikke statisk medlem kaldes for et objekt medlem, fordi det kun gælder i objektet.

```
class Min
{
     static int a;
     .....
}
```

Fig 6.9.3.2

Ved at kombinere `static` med `final`, kan man opnå at oprette en konstant der gælder for alle tilfælde af klassen, det kan se således ud:

```
class Min
{
     final static int a = 0;
     .....
}
```

Fig 6.9.3.3

Umiddelbart kan det virke overflødigt at erklære en konstant `static`, den bliver jo oprettet og tilskrevet når der oprettes et tilfælde af klassen. Fordelen er at man sparer plads i hukommelsen, fordi static kun allokeres én gang for alle tilfælde af klassen.

Fig 6.9.3.4 viser et eksempel på brugen af `static`.

```
// Filnavan = static1.java

class En_Klasse
{
        static int ObjektNummer = 0;
        En_Klasse()
        {
                ObjektNummer++;
                System.out.println("Dette er det " + ObjektNummer +
                " tilfælde af denne klasse");
        }
}

public class static1
{
        public static void main(String arg[])
        {
                En_Klasse a = new En_Klasse();
                En_Klasse b = new En_Klasse();
                En_Klasse c = new En_Klasse();
```

```
        }
}
```

```
C:\java static1
Dette er det 1 tilfælde af denne klasse
Dette er det 2 tilfælde af denne klasse
Dette er det 3 tilfælde af denne klasse

C:\
```

Fig 6.9.3.4

Hvis en metode erklæres som `static`, gælder der det samme som for variabler, der allekores kun plads til metoden én gang. Dette kan igen bruges til at spare plads i hukommelsen, hvis der skal erklæres mange tilfælde af en klasse.

Vi har ved flere lejligheder benyttet udtrykket

```
Math.random()
```

til at generere pseodo tilfældige tal i vores eksempler, uden først at at oprette et tilfælde af Math. Det kan kun lade sig gøre, fordi `random()` metoden i `Math` klassen er erklæret som `static`. Et andet sted hvor vi systematisk har brugt `static` er ved erklæring af main. Hvis du har glemt det vil kompileren straks brokke sig. Det skyldes, at den eneste måde Javaengine kan få fat i noget kode, er ved at den er klar til brug.

Derfor skal alle metoder i den klasse hvor main erklæres også erklæres som `static`. Hvis det ikke er tilfældet, så er der ingen kode som Java engine kan eksekvere. Kode oprettes med `new`, og det kan Javaengine ikke gøre af sig selv.

6.9.4 Konstruktøren og oprydning efter et objekt

Som alle andre metoder, kan konstruktøren også have en adgangskontrol. Ofte vil konstruktøren være erklæret public, men de andre niveauer kan også benyttes. Da konstruktøren er en speciel metode, er det vigigt at være sig bevidst om hvilken form for adgangskontrol man vælger at sætte op.

private	Dette niveau skal bruges med forsigtighed, idet ingen anden klasse kan oprette et tilfælde af en klasse, hvis konstruktør er erklæret private. Brugen af private erklærede konstruktører er ud over denne bogs rammer.
protected	Kun klasser der nedarver fra denne klasse, kan oprette et tilfælde af denne klasse.
public	Alle kan oprette et tilfælde af klassen
package	Kun medlemmer af den pakke hvor klassen er implementeret, kan oprette et tilfælde af klassen.

6.9.5 finalize

Java har en teknik der kaldes garbage collection. Det betyder, at når objekter og variabler ikke bruges mere, (går uden for rækkevidde) så vil garbage collectoren sørge for at rydde op, og derved frigive hukommelse.

Normalt kører garbage collectoren i det der kaldes asynkron tilstand, det betyder groft sagt, at den kører af og til når run-time systemet mener det er praktisk. Men den kan provokeres i gang af programmøren på flere måder, en af dem er med metoden gc fra Systemklassen. Vi kommer tilbage til den om lidt.

Inden et objekt destrueres kan det være en fordel at rydde lidt op først. I andre objektorienterede sprog gøres dette ved hjælp af en destruktør, der kaldes når objektet går ud af rækkevidde. I Java er destruktøren gemt for programmøren, så han ikke kan komme til den. Programmøren kan i stedet bruges `finalize` metoden.

En `finalize` metode overskriver Object klassen's `finalize` metode, den kaldes når garbage colectoren går i gang med at rydde op. En `finalize` metode erklæres på følgende måde:

```
protected void finalize() throws Throwable
```

I `finalize` metoden er det naturligt at udføre oprydning, selv om vi ikke har været inde på det endnu, kan det dreje sig om lukning af filer og netværksforbindelser.

I Fig 6.9.5.1 er vist et eksempel hvor der er implementeret en finalize metode i klassen `En_Klasse`. Metoden gør det modsatte ef konstruktøren (udskriver objektnummeret, og tæller det derefter en ned). Da `finalize` metoden ikke kaldes når objektet forsvinder ud af rækkevidde, men først når garbagekollekteren går i gang, fremprovokeres garbage collectionen med `System.gc()` kaldet i main.

Selv om garbage collectoren startes, kan der godt gå lidt tid inden den egentlig kommer i gang, derfor er der indlagt en dobbelt løkke. Denne løkke har kun til formål at lade lidt tid gå, således at vi kan se resultatet af `finalize` metoden.

Garbage collection kan bruge mærkbar tid, derfor kan det være en fordel at starte den på et tidspunkt hvor programmøren mener det er praktisk, fremfor selv at lade den vælge hvornår.Ved at bruge `gc()` metoden, kan vi altså bestemme tid og sted (omtrent) for garbage collectoren.

```
// Filnavn = static2.java


class En_Klasse
{
        static int ObjektNummer = 0;
        En_Klasse()
```

```
        {
                ObjektNummer++;
                System.out.println("Dette er det " + ObjektNummer +
                " tilf'lde af denne klasse");
        }

        protected void finalize() throws Throwable
        {
                System.out.println("Dette er det " + ObjektNummer +
                " tilfælde af denne klasse");
                ObjektNummer--;
        }
}


public class static2
{
        public static void main(String arg[])
        {

                En_Klasse a = new En_Klasse();
                En_Klasse b = new En_Klasse();
                En_Klasse c = new En_Klasse();

                System.gc();

                // Denne dobbeltløkke indlægger en forsinkelse for at
                // give tid til at Garbage Collecteren kan nå at starte.
                for (int i = 0; i < 32000; i++)
                        for (int j = 0; j < 10; j++)
                                ;
        }
}
```

```
C:\java static2
Dette er det 1 tilfælde af denne klasse
Dette er det 2 tilfælde af denne klasse
Dette er det 3 tilfælde af denne klasse
Dette er det 3 tilfælde af denne klasse
Dette er det 2 tilfælde af denne klasse
Dette er det 1 tilfælde af denne klasse
C:\
```

Fig 6.9.5.1

Opgave 6.9.1 Skriv et program der kan modtage et kontonummer, og et beløb. Den givne konto skal da opdateres med beløbet. Programmet skal indeholde klassen bank der skal indeholde metoderne

```
private void Opdater_Konto(float konto[],
        int KontoNummer, float beloeb)

public void Hent_Data(float konto[])
```

Metoden `Opdater_Konto` skal opdatere `konto`[`KontoNummer`], med det antal

kroner og ører der er angivet i beloeb. Metoden `Hent_Data` skal kaldes fra hovedprogrammet, og hente kontonummer og beløbet fra tastaturet.

Det maksimale antal kontonumre skal fremgå af en konstant, der har rækkevidde så den kan ses af hele klassen bank.

Opgave 6.9.2 Udvid klassen bank fra opgave 6.9.1 med en metode til at bestemme bankens samlede indestående

```
private float Samlet_Saldo(float konto[])
```

Opgave 6.9.3 Udvid klassen bank fra opgave 6.9.1 med en metode der foretager rentetilskrivning på samtlige konti når den bliver kaldt. Metoden skal se således ud.

```
private void Rente_Tilskrivning(float konto[],
float RenteSats)
```

Opgave 6.9.4 Udvid programmet fra opgave 6.9.3, således at der til hver konto hører et ejernavn.

Hint: Skift kontoarrayet ud med et objektarray,

Opgave 6.9.5.1 Skriv en klasse `Person_Klasse`. Klassen skal indeholde variablen personnavn, en metode til at indtaste navnet, og en metode til at slette navnet, og dermed en person. Klassen skal holde styr på hvor mange personer der er indtastet

Test klassen ved at oprette 10 tilfælde af den i main, og tast et antal navne ind.

Hint: Benyt en static int til at holde styr på hvor mange personer der er indtastet.

```
class Person_Klasse
{
        private String P_Navn;

        void Indtast_Navn()   {  ...   }
        void Slet_Navn()      {  ...   }
}
```

6.9.6 Vector klassen

Som et praktisk eksempel på klasse begrebet, vil vi bruge Vector klassen, der er implementeret i java.util pakken. Vector klassen er i stand til at styre alle operationer i en liste, det der i andre programmeringssprog kaldes en kædet liste. I Java version 1.2 er der nogle ændringer i Vector klassen, men det er blot udviddelser. Da version 1.2 ikke er officielt udgivet i skrivende stund, er

programmet testet i 1.2beta4 udgaven, hvilket fungerer helt problemfrit.

```
// Filnavn = vector1.java


import java.util.*;

class Data_Klasse
{
        String Lokal;

        void Indsaet_Data(Vector x)
           throws java.io.IOException
        {
                int ind;
                Data_Klasse a = new Data_Klasse(); // Skaber et nyt objekt

                a.Lokal = "";
                System.out.print("Indtast navn : ");
                while ((ind = System.in.read()) != 13)
                        a.Lokal += (char) ind;
                ind = System.in.read();            // Fjerner CR
                x.addElement(a);                   // Indsætter i liste
        }

        void Udskriv_Data()
        {
                System.out.println("Navn : " + Lokal);
        }
}

public class vector1
{
        public static void main(String arg[])
           throws java.io.IOException
        {
                int i;
                Data_Klasse a = new Data_Klasse();
                Vector Liste    = new Vector(); // Opretter en liste

                for (i = 0; i < 3; i++)
                        a.Indsaet_Data(Liste);

                System.out.println("Der er nu " + Liste.size() + " Elementer");

                for (i = 0; i < 3; i++)
                {
                        a = (Data_Klasse) Liste.elementAt(i);
                        a.Udskriv_Data();
                }
        }
}
```

C:\java vector1
Indtast navn : Søren
Indtast navn : Peter
Indtast navn : Trine
Der er nu 3 elementer
Navn : Søren
Navn : Peter
Navn : Trine

C:\

Fig 6.9.6.1

I dette eksempel vil vi kort kommenter et par metoder i `Vector` klassen. For yderlige informationer om `Vector` klassen bør du downloade dokumentationen fra www.javasoft.com. Det er ganske gratis.

Fig 6.9.6.1 kan deles op i to dele der er næsten uafhængige af hinanden, `Data_Klasse` og `vector1` klassen. Vi kan roligt starte i `vector1` klassen.

Til at starte med erklærer vi et tilfælde af `Data_Klasse`. Dette tilfælde benytter vi til at kalde et objekt, der i samarbejde med `Vector` klassen laver alt arbejdet for os. Inden vi kan starte skal vi dog have oprettet et tilfælde af `Vector` klassen, dette kalder vi for `Liste`, fordi vi vil konstruere en liste.

Derefter løber vi tre gange rundt i en `for` løkken (Fra nul til tre), for at taste ind. På nuværende tidspunkt behøver vi ikke bekymre os om hvordan det sker, vi kan bare se at der er en metode i objektet `a` ved navn `Indsaet_Data`, der gør alt det grove arbejde for os.

Når det er gjort kaldes metoden `size()` i `Vector` klassen, for at oplyse brugeren om hvor mange der er tastet ind.

Til sidst tager vi fat i elementerne i listen efter deres placering i listen, numer et først. Her bruges igen en metode fra `Vector` klassen til at udtage det i´de element fra listen, og via en cast, placere en reference til objektet i listen i objektet `a`. Herefter slutter vi af med at kalde `a´s` metode til udskrivning. Det er nødvendigt at udføre en typekonvertering (cast) når noget hentes ud af Vector klassen, da den jo kan aflevere alle slags objekter.

Hele main metoden ligner et stykke pseudo kode, men den er ganske funktionel. Det der nok springer mest i øjnende er, at `a` kun bliver oprettet én gang i main. Det er ikke nødvendig at oprette flere, for hvis vi kigger i `Indsaet_Data` klassen, så opretter den et objekt af rette type for os. I main har vi altså et to objekter, et der opretter nye objekter (`a`), og et hvor vi kan opbevare lige så mange objekter som computeren har RAM og virtuel RAM til at rumme (`Liste`)

`Data_Klassen` har to metoder, hvoraf `Udskriv_Data` nok må siges at være triviel, så lad os kigge lidt på `Indsaet_Data` metoden.

Bemærk at der i klassen oprettes et nyt tilfælde af klassen, dette er ikke det samme tilfælde som det vi arbejder i, men et ganske andet. Det er ej heller det samme som evt. blev oprettet sidste gang vi kaldte metoden, for metoden har været uden for rækkevidde, (uden for scope) og har derfor ingen reference til det “gamle” tilfælde af `a`.

Lige før `Indsaet_Data` forlades, kaldes `addElement` metoden fra `Vector` klassen, dette medfører at objektet `a` i `Data_Klasse` bliver refereret til fra `Vector` klassen, som vi oprettede et tilfælde af i main.

6.10 Opgaver

Opgave 6.10.1 Tilpas programmet fra Fig 6.9.6.1 således at man hele tiden, via en metode i klassen `Data_Klasse`, kan spørge på hvor mange elementer der er oprettet.

Opgave 6.10.2 Udvid Opgave 6.10.1 med en `finalize` metode, der tæller ned ned når et objekt destrueres.
Hint: Se removeElement(Object) og removeElementAt(int) i Vector klassen.

Opgave 6.10.3 Tilpas programmet fra Fig 6.9.6.1 således at al kontakt med brugeren (brugerinterfacet) placeres i en klasse for sig.

Opgave 6.10.4 Tilpas programmet fra Fig 6.9.6.1 således at Listen kommer ind i `Data_Klasse` klassen.

7 Filhåndtering

Inden vi går i gang med filhåndteringen i Java, skal læseren være opmærksom på, at på grund af sikkerheden i en applet, er det ikke sikkert at en applet får lov til at skrive til, eller læse fra en fil. I den forbindelse skal der skelnes mellem Java applikationer, og Java appletter. Java applikationer har en static main metode, appletter har ikke. Appletter kører normalt i en browser, og det vil være normalt at browserens sikkerhedsopsætning ikke tillader filadgang. Java applikationer har ingen restriktioner på filadgang, med mindre applikationen køres i en browser.

Dette kan justeres på flere måder, men det er uden for denne bogs rammer at komme nærmere ind på det. Interesserede kan undersøge begrebet “Security Manager”.

7.1 Filer

Ordet fil kommer af det engelske file, der betyder arkiv. I datatermonologien er en fil en samling af bits, placeret på et lagermedie. Lagermediet kan være bånd, diskette, harddisk, CD-ROM og lignende.

Set fra en programmørs synsvinkel, er en fil et sted hvor man gemmer data til senere brug. Når en fil indlæses sker det ved at man modtager en strøm af data (stream) fra lagermediet, indtil hele filen, eller den del man ønsker, er indlæst i computerens RAM. Det omvendte er tilfældet når man skriver til en fil, i så fald går strømmen fra RAM’en til disken.

7.2 Input fra fil

Der er flere måder hvorpå man i Java kan læse fra en fil. Ved at oprette et tilfælde af klasserne `FileInputStream` og `DataInputStream`, kan filens data hentes ind som en datastrøm. Dette kræver at java.io pakken er importeret. Fig 7.2.1 viser et program der udskriver indholdet af en fil på skærmen.

Programmet laes_fil.java på Fig 7.2.1 oprettes et objekt af typen `FileInputStream` ved navn `IndFil`. Dette objekt bliver tilknyttet til den fil der ønskes læst (laes_fil.java). Filnavnet overføres til objektets konstruktør.

Selve filen bliver udskrevet i `while` løkken. Årsagen til at alle linieskift udskrives korrekt er, at de læses ind fra filen og skrives til skærmen, hvilket medfører et linieskift. End Of File tegnet vil blive indlæst i variablen `Ind`, men vil ikke blive sendt til skærmen, da netop i det tilfælde ophører betingelsen i `while` løkken med at være sand.

```
// Filnavn = laes_fil.java
// Dette program udskriver sig selv på skærmen, forudsat at
// kildeteksten ligger i samme bibliotek som den kompilede.

import java.io.*;

public class laes_fil
{
    public static void main(String arg[])
    {
        final int EOF = -1;
        try
        {
            FileInputStream IndFil = new FileInputStream("laes_fil.java");
            DataInputStream IndStroem = new DataInputStream(IndFil);
            int Ind;

            while ( (Ind = IndStroem.read()) != EOF)
                System.out.print( (char) Ind);
            IndFil.close();
        }
         catch (IOException Fejl)
         {
            System.out.println("Fejlkode : " + Fejl);
        }
    }
}
```

Dette program udskriver sig selv under kørslen.

Fig 7.2.1

Inden blokken forlades bør man lukke de filer man har åbnet, dette gøres med `close()` metoden fra `FileInputStream`.

Der skal oprettes en `try` blok rundt om al filhåndtering, ellers vil kompileringen ikke blive fuldført.

Opgave 7.2.1 Prøv at ændre filnavnet i `FileInputStream` til et filnavn der ikke eksisterer, hvad sker der?

Opgave 7.2.2 Omskriv programmet laes_fil således, at filnavnet overføres fra kommandolinien.

7.3 Output til fil

Hvis man ønsker at skrive til en fil, skal man oprette et tilfælde af klasserne `FileOutputStream` og `DataOutputStream`. Fig 7.3.1 viser et enkelt filskrivningsprogram. Som ved fillæsningsprogrammet, skal java.io pakken importeres.

Programmet virker i store træk som Fig 7.2.1. Først oprettes objektet `UdFil` som et tilfælde af `FileOutputStream`, filnavnet overføres til konstruktøren. Derefter skabes der et objekt af typen `DataOutputStream`.

Når programmet kører, beder det høfligt brugeren om at begynde indtastningen, herefter kan brugeren taste de tegn ind han ønsker. Når brugeren trykker på enter stoppes `while` løkken, og selv om variablen Ind indeholder tegnet CR, (carrige return = enter) vil det ikke blive skrevet i filen, da udtrykket i `while` løkken ikke er sandt.

```
// Filnavn = skriv_fil.java
// Dette program sender input fra tastaturet til en fil ved navn X_file
//Hvis der findes en fil ved nmavn X_file i arbejdsbiblioteket, vil
// den blive overskrevet.

import java.io.*;

public class skriv_fil {
    public static void main(String arg[])    {
        final int CR = 13;
        try        {
            FileOutputStream UdFil = new FileOutputStream("X_file");
            DataOutputStream UdStroem = new DataOutputStream(UdFil);
            int Ind;

            System.out.println("Begynd indtastning");
            while ( (Ind = System.in.read()) != CR)
                UdStroem.write( Ind);
            UdFil.close();
        }
        catch (IOException Fejl)
        {
            System.out.println("Fejlkode : " + Fejl);
        }
    }
}
```

C:\ java skriv_fil
Begynd indtastning
abcdefg↵

C:\

Herefter vil der stå abcdefg i filen ved navn X_file. Kontroller ved at skrive TYPE X_FILE

Fig 7.3.1

Det er specielt vigtigt at lukke en fil der er skrevet til, da filen først bliver opdateret, og dens indhold dermed er konsistent, efter den er blevet lukket.

Som ved læseprogrammet skal alle filfunktioner foregå inden i en `try` blok.

Opgave 7.3.1 Prøv at ændre filnavnet i `FileOutputStream` til et filnavn der ikke eksisterer, hvad sker der?

Opgave 7.3.2 Omskriv programmet skriv_fil således, at filnavnet overføres fra kommandolinien.

7.4 Anvendelighed

For at gøre filadgangen nemmere, kan det være passende at skabe sin egen filklasse, således at man nemt kan hente indholdet af en fil. Fig 7.4.1 giver et eksempel på 1. skridt i opbygningen af en filklasse.

På Fig 7.4.1 er vist et eksempel hvor en filklasse udfører alt det "grove" arbejde. Brugeren af klassen skal blot kalde en metode i klassen med argument for det filnavn han ønsker hentet, argument til at indeholde data fra filen, samt en `int` der oplyser hvor meget der maksimalt må indlæses, grundet arrayets valgte størrelse.

Hvis dette kald går godt, returneres filens længde, og filens indhold. Herved har brugeren af klassen, (programmøren) nået et højere abstraktionsniveau. Sagt med andre ord, han har gjort det nemmere for sig selv næste gang han skal bruge en fil.

Dette program virker set fra brugeren, på samme måde som Fig 7.2.1.

Programmet startes med en parameter, denne skal være navnet på en tekstfil. Hvis det ikke er en tekstfil vil skærmen opføre sig besynderligt. Hvis filnavnet ikke findes, fanger catch blokken fejlen, og udskriver en fejlmeddelelse.

```
// Filnavn = fil1.java

import java.io.*;

class Fil_Klasse
{
    final int EOF = -1;

    public int Hent_Fil(String Filnavn, int Data[], final int MAX)
    {
        int i = 0;
        try
        {
            FileInputStream IndFil = new FileInputStream(Filnavn);
            DataInputStream Stroem = new DataInputStream(IndFil);

            while( ((Data[i++] = Stroem.read()) != EOF) && (i < MAX) )
                                ;
                Stroem.close();
        }
        catch (IOException Fejl)
        {
            System.out.println("Fejlkode : " + Fejl);
        }
        return --i;
    }
}
```

```
public class fil1
{
    static final int MAX = 1000;

    public static void main(String arg[])
    {
        int i,Graense;
        int FilData[]  = new int[MAX];
        Fil_Klasse Fil = new Fil_Klasse();

        Graense = Fil.Hent_Fil(arg[0], FilData, MAX);
        for (i = 0; i < Graense; i++)
            System.out.print( (char) FilData[i]);
    }
}
```

Hvis programmet startes således:
C:\java fil1 C:\autoexec.bat
Så vil indholdet af autoexec.bat filen blive skrevet på skærmen, forudsat den ligger i roden af drev C:

Fig 7.4.1

7.5 Opgaver

Opgave 7.5.1 Udvid Fil_Klasse fra Fig 7.4.1 med en metode der skriver indeholdet af et array af char, til en fil. Filnavnet skal overfres som argument til metoden.

Opgave 7.5.2 Skriv et program, der ved brug af Fil_Klasse fra opgave 7.4.1 kopierer fra en fil, til en anden. Begge filnavne skal hentes fra kommandolinien.

Opgave 7.5.3 Tilpas opgave 5.1.4 således at det der skal kryptograferes kommer fra en fil, og resultatet gemmes i en fil.

Opgave 7.5.4 Skriv et program der dekryptograferer en fil oprettet af opgave 7.4.3.

Opgave 7.5.5 Tilpas opgave 5.1.6 således at det der skal dekryptograferes kommer fra en fil.

Opgave 7.5.6 Skriv et program der analyserer indholdet af en tekstfil. Analysen skal oplyse hyppigheden af de enkelte bogstaver i en fil.

Opgave 7.5.7 Kodebrydere fandt tidligt ud af, at hvert sprog har sin frekvensfordeling af bogstaver. Tilpas derfor programmet fra Opgave 7.5.6 og 7.5.5 således, at programmet først læser en fil på det sprog hvor en kode siden hen skal brydes. Resultatet er en hyppighedsfordeling af bogstaver for det givne sprog. Derefter skal den kryptograferede fil med det ukendte offset indlæses. Koden skal derefter brydes automatisk, ved at prøve alle kombinationsmuligheder af, og udskrive den hvor fordelingen af bogstaver passer bedst med sprogets fordeling af bogstaver.