

# Java-kurs

Andreas Knudsen <andreakn@idi.ntnu.no>  
Nils Grimsmo <nilsgri@idi.ntnu.no>

9th October 2003

(Dette blir lagt ut på nett, du trenger ikke ta notater.)

## 1 Motivasjon

For de som ikke går å data:

- Programmering er en viktig del av faget
- Programmeringsoppgaver kan komme på eksamen
- Hvis du skal kunne bruke det du har lært i faget siden, må du enten kunne programmere selv, eller leie inn noen som kan det.

For de som går på data (og lignende):

- De er ikke for sent for de som ikke fikk kontroll på programmeringen i SIF8005 objekt-orientert programmering.
- Obligatoriske fag med programmering:
  - Systemutvikling
  - Databaser
  - Kommunikasjon, tjenester og nett
  - Programmeringsspråk
  - Logikk
  - Operativsystemer
  - Kunnskapssystemer
- "Du trenger ikke kunne programmere for å begynne på data" betyr ikke det samme som "Du trenger ikke kunne programmere når du er ferdig på data."
- Denne høsten er en ypperlig sjans til å lære seg å programmere, lærer du det nå, vil du ha mye igjen for det i resten av studiet.

## 2 Hvordan lære seg å programmere?

Programmering kan IKKE læres på forelesning. Det læres ved praktisk øving.

Det er alltid en sammenheng mellom det å mestre noe, og det å synes det er gøy.

Det er to tilnærmelser til denne problemstillingen:

- Du kan tvinge deg selv til å lære noe. Når du da kan det, synes du enten det er gøy, eller så hater du det :)
- Du kan prøve å få det du skal lære til å bli morsommere, og slik bli mer motivert til å lære.

En kombinasjon av disse to er nok det som fungerer i praksis.

Tips: Sett deg ned en kveld eller en helg sammen med noen som kan omtrent like mye, og prøv å løse en del oppgaver. (Gjem fjernkontrollen, ikke noe Cæsar!)

Dersom du kjenner noen som kan mer enn deg, og har lyst til å lære bort, er det gull. Man kan for eksempel bake en kake, eller spandere en pils eller et glass rødvin.

Et annet hett tips er å lage seg noen oppgaver *selv*. Dette er mer motiverende. Lag noen programmer som finner ut noe du lurer på. Eventuelt kan du late som om du lurer på det :)

Det er en del forslag til oppgaver i dette foilsettet.

## 3 Grunnleggende ferdigheter

### 3.1 Primitive typer

Uttrykket "flate variable" brukes om primitive typer. Primitive typer er atomiske, eller ikke sammensatte, typer.

Primitive typer er f.eks.:

- char (tall mellom 0 og 255, eller en bokstav)
- int (heltall)
- long (stort heltall)
- float (flyttall)
- double (flyttall)
- boolean (sannhetsverdi)

Variabler deklarerer på denne måten:

```
int mittTall;  
double mittTall2;  
boolean erOk;
```

Variabler kan også initieres (tilordnes en verdi) samtidig med at de deklarerer.

```
int mittTall = 13;  
double mittTall2 = 2.7182818284590451;  
boolean erOk = false;
```

Strenger er ikke primitive typer i Java, men objekter. Likevel kan de brukes omtrent som primitive typer. Eksempel:

```
String hei = "hei";  
hei = "hallo";
```

## 3.2 Enkle uttrykk og tilordning

Tilordning gjøres i Java med operatoren '='. Man skriver:

```
variabel = uttrykk;
```

Eksempel:

```
int mittTall = 2;
System.out.println(mittTall);
mittTall = 3;
System.out.println(mittTall);
```

Matematiske utrykk kan formuleres i Java.

Eksempel:

```
// 28% skatt
double skattesats = 28.0 / 100.0;
// tusen kroner i uka i ett år
double inntekt = 1000 * 52;
// regner ut skatten
double skatt = inntekt * skattesats;
// skriver ut skatten
System.out.println("skatt betalt:" + skatt);
```

Som sett her, kan strenger kan konkateneres med operatoren '+'. Eksempel:

```
String en = "hei";
en = en + ", på deg!";
String to = "hva heter du?";
String setning = en + " " + to;
```

### 3.3 Array

Array er lister som er sammensatt av elementer. Her er et eksempel på bruk av array:

```
// deklarerer 'tabell' til å være en liste av int'er
int[] tabell;
// tilordner en liste av 10 int'er til 'tabell'
tabell = new int[10];
// tilordner tallet 13 til det nullte elementet
tabell[0] = 13;
// femte elementet i tabell har nå verdien 23
tabell[5] = tabell[0] + 10;
```

'tabell' ser nå slik ut:

```
{13, 0, 0, 0, 0, 23, 0, 0, 0, 0}
```

Tabeller er i Java (og de fleste andre språk) indeksert fra null. Det vil si at det fremste elementet er element nr 0. Hvis det er 10 elementer, er det siste elementet da nr 9.

Pseudo-koden i boken indekserer arrayer fra 1.

Java initierer verdiene i arrayer til 0. Ikke alle programmeringsspråk gjør dette.

### 3.4 Matriser

Matriser er todimensjonale arrayer. De kan sees på som en array av array'er, akkurat som at en matrise i matematikken kan sees på som en vektor av vektorer.

Her er et eksempel på bruk:

```
int[] [] matrise = new int[3][3];
matrise[0][2] = 2;
matrise[1][1] = 1;
matrise[2][0] = 5 + matrise[0][2];
```

'matrise' ser nå slik ut:

```
{{0, 0, 2},
 {0, 1, 0},
 {7, 0, 0}}
```

## 4 Rammeverk

For å få gjort ting i Java trenger vi et lite rammeverk rundt programmet, eller et skall, akkurat som i programmeringsøvingene.

Her er et lite rammeverk i Java, som vi bruker i resten av gjennomgangen

```
class Klasse {  
  
    public static void f(int i) {  
        System.out.println("f() fikk " + i +  
            " som innparameter");  
    }  
  
    public static void main(String[] args) {  
        f(1);  
    }  
}
```

Inn i dette lille skallet skal vi putte programmene våre. Metoden 'main' blir automatisk kjørt av Java når du starter programmet.

## 5 Kontrollflyt

Kontrollflyt er det som gjør at programmer kan utføre oppgaver på varierende inndata.

Vi har tre generelle begreper:

- Sekvens
- Betingelser
- Løkker

### 5.1 Sekvens

At programmer blir sekvensielt utført betyr ganske enkelt at programmet blir utført linje for linje, med mindre kontrollstrukturer tilsier noe annet.

Eksempel:

```
// denne linjen blir utført først...
int x = 2 + 1;
// ...så denne...
int y = 13;
// ...og denne
x = y * x * x;
```

## 5.2 Betingelser

Dette er betinget utførelse, også kjent som "if-statements".

Eksempel:

```
if (x < 5) {  
    System.out.println("x var mindre enn 5");  
}
```

I Java kan du også lage mer utførlige if-løkker. Her har du et litt større eksempel satt inn i et program:

```
class Klasse {  
  
    public static void f(int x) {  
        int y = (x-3);  
        if (x < 5) {  
            System.out.println("x var mindre enn 5");  
        }  
        else if (y > 10 && x == 8) {  
            System.out.println("x ikke mindre enn 5,");  
            System.out.println("y større enn 10, x lik 8");  
        }  
        else {  
            System.out.println("ingen av tilfellene over");  
        }  
    }  
  
    public static void main(String[] args) {  
        f(13);  
    }  
}
```

Hva vil dette programmet skrive ut? (Gå igjennom sekvensielt)

Betingelsene som står inni hodet til if-løkkene må være bolske uttrykk, eller uttrykk som har en sannhetsverdi. Her er noen eksempler på bolske uttrykk

- $x < 5$
- $y == 3$
- $x < 10 \ \&\& \ x > 10$

Akkurat som man har aritmetiske operatører på tall (+, -, \* og /), har man logiske operatører på sannhetsverdier. Hvis b1 og b2 er sannhetsverdier, kan man si:

- $b1 \ \&\& \ b2$  ("and", sant hvis både b1 og b2 er sann)
- $b1 \ || \ b2$  ("or", sant hvis minst en av b1 og b2 er sann)
- $!b1$  ("not", sant hvis b1 er usann)
- $b1 \ \hat{\ } \ b2$  ("exclusive or", sant hvis bare en av b1 og b2 er sann)

Bolske verdier kan også tilordnes til variable:

```
int x = 10;
boolean minBool = true;
minBool = minBool \&\& x < 10;
if (minBool) {
    System.out.println("hurra!");
}
else {
    System.out.println("D'OH!");
}
```

## 5.3 Løkker

Løkker er kontrollstrukturer som gjør at instruksjoner kan repeteres.

I java har vi:

- while
- do-while
- for

### 5.3.1 while

while har strukturen:

```
while (sannhetsverdi) {  
    // utfør dette  
}
```

En kan oversette 'while' med 'så lenge'. "Så lenge det som sannhetsverdien som står i hodet til løkken har verdien true, utfør det som står inni kroppen til løkken.

eksempel:

```
int x = 0;  
while (x < 10) {  
    System.out.println(x);  
    x = x + 1;  
}
```

dette lille programmet skriver ut tallene fra og med 0 til og med 9.

### 5.3.2 do-while

'do-while' er omtrent det samme som 'while', bortsett fra at betingelsen blir sjekket på slutten av hver runde i løkken. Det vil si at løkken blir kjørt minst en gang. Her er et eksempel:

```
class Klasse {  
  
    public static void f() {  
        int x = 0;  
        int y = 10;  
        do {  
            System.out.println(x + "," + y);  
            x = x + 1;  
            y = y - 1;  
        } while (x > 0 && x != y);  
    }  
  
    public static void main(String[] args) {  
        f(13);  
    }  
}
```

Hvor mange "runder" blir kjørt i dette programmet?

### 5.3.3 for

for-løkker er litt mer avanserte while-løkker. Alle for-løkker kan enkelt skrives om til while-løkker. Her er et eksempel på en for-løkke:

```
for (int x = 0; x < 10; x++) {  
    System.out.println(x);  
}
```

'x++' betyr det samme som 'x = x + 1'

I hodet til for-løkken er det tre uttrykk, adskilt med semikolon.

- Det første blir utført før løkken
- Den andre er en betingelsen som må være sann for at løkken skal utføres
- Det tredje uttrykket blir utført på slutten av hver runde i løkken

Med andre ord kan en for-løkke med uttrykkene u1, u2, og u3 i hodet skrives om til en while-løkke på denne måten:

```
for (u1; u2; u3) {  
    // noe  
}
```

kan skrives:

```
u1;  
while (u2) {  
    // noe  
    u3;  
}
```

Her er et eksempel på et program som skriver ut alle elementene i en array:

```
class Klasse {  
  
    public static void f(int[] minArray) {  
        for (int i = 0; i < minArray.length; i++) {  
            System.out.println(minArray[i]);  
        }  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {2,6,8,4,3,5};  
        f(arr);  
    }  
}
```

Notasjonen '{2,6,8,4,3,5}' er en spesialnotasjon for initiering av arrayer.

## 5.4 Oppgaver

Her er et lite sett med oppgaver du kan prøve å løse:

Lag et program som:

- Finner det minste elementet i et array
- Finner det største og det minst elementet i en array
- Finner summen av alle elementene i et array
- Finner ut hvilket av to arrayer som har størst sum
- Finner prikk-produktet av to vektorer (arrayer)

Her har vi løst den første oppgaven:

```
class Klasse {  
  
    public static void f(int[] minArray) {  
        int minste = Integer.MAX\_VALUE;  
        for (int i = 0; i < minArray.length; i++) {  
            if (minArray[i] < minste) {  
                minste = minArray[i];  
            }  
        }  
        System.out.println("Minste verdi: " + minste);  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {2,6,8,4,3,5};  
        f(arr);  
    }  
}
```

## 6 Abstraksjoner

For at det skal bli lett å uttrykke ting, behøver vi litt mer en bare kondisjoner og løkker.

Vi har følgende abstraksjoner, som vi skal forklare i tur og orden:

- Metoder
- Klasser og objekter

### 6.1 Metoder (funksjoner)

Vi har allerede sett et eksempel på en funksjon i programmene våre. Den het 'f'. I java kalles funksjoner for metoder, og har følgende syntax:

```
public static void f(innparametre) {  
    // kropp  
}
```

De to første ordene har med objektorientering å gjøre, og vi skal forklare disse siden.

Det tredje ordet, 'void', er returtypen til metoden. Void betyr "ingenting", eller "en tom verdi", så denne metoden returnerer ikke noe.

En metode kan returnere en verdi. Her er et eksempel:

```
public static int ganger2(int x) {  
    int svar = x * 2;  
    return svar;  
}
```

Denne kunne også vært skrevet:

```
public static int ganger2(int x) {
    return x * 2;
}
```

Denne metoden returnerer en verdi av typen int.

Retur-typen til en metode kan være void (tom), en primitiv type (int, boolean, double osv), eller objekter.

Her er et litt mer komplisert eksempel:

```
class Klasse {

    public static double skatt(double g, double sats) {
        return g * skattesats;
    }

    public static double netto(double brutto) {
        return brutto - skatt(brutto, 0.28);
    }

    public static void main(String[] args) {
        System.out.println("Netto: " + netto(1000000));
    }
}
```

Fra main-metoden kalles metoden netto(), med parameteret 1000000. Når programmet kommer inn i dit, har variabelen 'brutto' verdien 1000000. Herfra blir så videre metoden skatt() kalt, med parameterene 'grunnlag' = 1000000, og 'skattesats' = 0.28. skatt() beregner så hvor mye skatt som skal betales, og returnerer dette. Kontrollflyten er nå tilbake i netto(), som returnerer 'brutto' minus skatten beregnet av skatt(). Så kommer kontrollen tilbake til main(), som skriver ut resultatet.

### 6.1.1 Rekursjon

Rekursjon vil si at en metode kaller på seg selv, eller at flere metoder kaller på hverandre. Et eksempel på det siste er at f() kaller på g() som igjen kaller på f().

Rekusjon brukes for å elegant kunne uttrykke løsninger på en enkel måte. Her er et eksempel som ikke er så elegant:

```
class Klasse {  
  
    public static void f(int x) {  
        System.out.println(x);  
        if (x < 10) {  
            f(x + 1);  
        }  
        System.out.println(x);  
    }  
  
    public static void main(String[] args) {  
        f(0);  
    }  
}
```

Dette programmet skriver ut tallene fra og med 0 til og med 10. Dette gjøres enklest med en løkke.

Noe som er svært viktig å forstå med rekusjon, er at når en metode kalles rekursivt, er det flere "instansieringer" av metoden. Variablene som er lokale til metoden (her 'x'), finnes det en av for hver instansiering av metoden. Først blir det kalt på f(0). "Instansen" f(0) finnes fremdeles mens f(1) blir kalt. Vi kan tegne dette slik:

```
f(0) -->
  x = 0
  f(1) -->
    x = 1
    f(2) -->
      x = 2
      f(3) -->
        x = 3
        f(4) ...
      <--
      x = 2
    <--
    x = 1
  <--
  x = 0
<--
```

'-->' betyr her metode-kall, '<--' betyr at et kall returnerer.

Hvis du ser på når metoden blir kalt for første gang, er variabelen  $x = 0$ . Hvis du ser på linjen etter der hvor kallet f(1) returnerer, er x fremdeles 0. Det er en instans av variabelen x for hvert kall til metoden.

Her er et klassisk, og litt mer interessant eksempel på rekusjon, fibonacci-tallene:

```

class Klasse {

    public static int fib(int x) {
        if (x == 1 || x == 2) {
            return 1;
        }
        else {
            return fib(x-1) + fib(x-2);
        }
    }

    public static void fibonacci(int n) {
        System.out.println("fib(" + n + ") er " + fib(n));
    }

    public static void main(String[] args) {
        for (int i = 1; i < 10; i++) {
            fibonacci(i);
        }
    }
}

```

Fibonacci-tallene er definert slik at det første og det andre er 1, mens resten av tallene er definert slik at hvert tall er summen av de to foregående.

Prøv å gå igjennom denne rekursjonen for  $i = 2, 3$  og  $4$ .

## 6.1.2 Oppgaver

Her er noen tips til oppgaver du kan prøve å gjøre:

- Gjør om noen av løsningene du gjorde under kapitlet kontrollflyt til å være metoder som returnerer noe.
- Lag en metode som finner summen av alle elementene i en tabell ved hjelp av rekursjon.
- Lag en metode som finner ut om elementene i et array har sam-merkefølge lest forlengs og baklengs ("agnes i senga" eller {1, 2, 4, 1, 4, 2, 1}). Dette gjøres lettest uten rekursjon.

## 6.2 Klasser og objekter

Klasser og objekter er grunnleggende begreper i objektorientert programmering.

En klasse kan sees på som en oppskrift for objekter.

Her er et eksempel på bruk av klasser og objekter:

klassen OO:

```
import java.util.ArrayList;

class OO {

    private ArrayList nodes = new ArrayList();

    public void addNode(Node n) {
        nodes.add(n);
    }

    public Node getNode(int i) {
        return (Node)nodes.get(i);
    }

    public int numNodes() {
        return nodes.size();
    }

    public static void main(String[] args) {
        OO oo = new OO();
        oo.addNode(new Node(10));
        Node node = new Node(13);
        oo.addNode(node);
        for (int i = 0; i < 5; i++) {
```

```

        oo.addNode(new Node(i));
    }
    System.out.println("Har følgende noder:");
    for (int i = 0; i < oo.numNodes(); i++) {
        Node n = oo.getNode(i);
        System.out.println(i + ": " + n.getValue());
    }
}
}

```

Klassen Node (som brukes i OO):

```

public class Node {

    private int value;

    public Node(int val) {
        value = val;
    }

    public int getValue() {
        return value;
    }
}

```

For å kjøre dette må klassen OO legges i en fil som heter "OO.java", og klassen Node i en fil som heter "Node.java".

Kompiler med kommandoen: "javac OO.java Node.java" Kjør med kommandoen: "java OO"

Det er mange ting å kommentere her:

### 6.2.1 Attributter og metoder

Som du ser har klassene flere metoder, som også vist i tidligere eksempler. Du kan kalle på en metode tilhørende et objekt med syntaksen:

```
objektnavn.metodenavn(parametre)
```

Eksempel:

```
oo.getNode(0)
```

Objektene har også tilhørende attributter, eller objektvariable. I eksempelet over er 'nodes' en attributt som tilhører objekter av typen OO. Objekter av typen Node har en attributt 'value' av typen int.

### 6.2.2 Klasser fra standardbiblioteket

I standardbiblioteket til Java finnes en lang rekke av klasser som kan brukes akkurat som klasser du lager selv. I eksempelet er klassen ArrayList brukt. For å kunne bruke denne har vi linjen "import java.util.ArrayList;" først i programmet. Det er viktig å kunne finne frem til nyttige klasser i biblioteket slik at du slipper å finne opp hjulet hver gang.

Det viktigste du bør gjøre hvis du vil sette deg litt inn i java er å sjekke det ferdiglagde klassebiblioteket som leveres sammen med Java: Du finner en fullstendig oversikt på Sun sine API-sider:

<http://java.sun.com/j2se/1.4.2/docs/api/index.html>

(link finnes fra fagsiden under eksterne ressurser)

'nodes' i klassen OO er en attributt av typen ArrayList.

### 6.2.3 Instansiering

Se på linjene

```
OO oo = new OO();
```

og

```
Node node = new Node(13);
```

Dette er instansiering av objekter av typene OO og Node. 'oo' og 'node' er nå variable, og kan brukes mye på samme måte som primitive typer, angående tilordning osv.

### 6.2.4 Objekter som parametre og retur-typer

Som du ser kan objekter brukes som parametre, f.eks. i kallet

```
oo.addNode(new Node(10));
```

De kan også returneres fra metoder:

```
public Node getNode(int i) {  
    return (Node)nodes.get(i);  
}
```

### 6.2.5 Static vs dynamic

Som du kanskje har lagt merke til står ikke ordet "static" foran i hodet til metodene. Dette betyr at de hører til objektene, og ikke til klassen.

Hvis en attributt eller metode er definert som statisk i en klasse, betyr dette at alle objektene som er instanser av klassen deler denne attributten eller metoden.

Her er en omskriving av klassen OO med statiske attributter og metoder:

```
import java.util.ArrayList;

class OO {
    private static ArrayList nodes = new ArrayList();

    public static void addNode(Node n) {
        nodes.add(n);
    }

    public static Node getNode(int i) {
        return (Node)nodes.get(i);
    }

    public int numNodes() {
        return nodes.size();
    }

    public static void main(String[] args) {
        OO oo = new OO();
        oo.addNode(new Node(9));
        oo.addNode(new Node(10));
        addNode(new Node(8));
        Node node = new Node(13);
        addNode(node);
        for (int i = 0; i < 5; i++) {
            addNode(new Node(i));
        }
        System.out.println("Har følgende noder:");
        for (int i = 0; i < oo.numNodes(); i++) {
            Node n = getNode(i);
            System.out.println(i + ": " + n.getValue());
        }
    }
}
```

Legg merke til følgende:

- Vi kan fremdeles lage objekter av typen OO, selv om alle attributtene til klassen er statiske ("OO oo = new OO()")
- Vi kan kalle på metoder på klassen ("OO.addNode(new Node(9))")
- Vi kan kalle på statiske metoder på et objekt, selv om metoden tilhører klassen, og ikke objektet ("oo.addNode(new Node(10))")
- Vi kan unlate å spesifisere hvilken klasse vi kaller på en metode i, gitt at kallet kommer fra den samme klassen ("addNode(new Node(8))" er det samme som "OO.addNode(new Node(8))")
- En ikke-statisk metode kan bruke statiske attributter og metoder:

```
public int numNodes() {  
    return nodes.size();  
}
```

Dette er fordi 'nodes' tilhører klassen, og metoden i objektet da entydig kan bestemme hvilken variabel 'nodes' det dreier seg om.

- En statisk metode kan ikke bruke ikke-statiske attributter og metoder, uten å spesifisere hvilke objekter disse skulle tilhøre. Grunnen til dette er at man ikke da kan vite hvilket objekt man skal se på. Her er et eksempel på dette:

```
class OO {  
  
    private ArrayList nodes = new ArrayList();  
  
    public static void addNode(Node n) {  
        nodes.add(n);  
    }  
}
```

```
public static void main(String[] args) {  
    OO oo1 = new OO();  
    OO oo2 = new OO();  
    addNode(new Node(8));  
}  
}
```

Her ser vi at det lages to objekter av typen OO. I den statiske metoden addNode(), har vi ingen mulighet til å vite om den nye noden skal inn i nodelisten til 'oo1' eller 'oo2'.

### 6.3 Referanser

En ting som må presiseres er at variable som ikke er av primitive typer er referanser til objekter, eller pekere. Dette vises best ved et eksempel:

```
Node n1 = New Node(13);  
Node n2 = n1;  
Node n3 = n2;
```

Vi har nå tre variable, 'n1', 'n2' og 'n3', men vi har bare ett objekt av typen Node. 'n1' er en referanse til et Node-objekt. Her er enda et eksempel, som bruker klassen ArrayList fra standardbiblioteket:

```
import java.util.ArrayList;

class Klasse {

    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(new Node(13));
        ArrayList a2 = a1;
        a2.add(new Node(14));
        System.out.println(a1.size());
        System.out.println(a2.size());
        a1.remove(0);
        System.out.println(a1.size());
        System.out.println(a2.size());
    }
}
```

Dette programmet skriver ut:

```
2
2
1
1
```

Av dette kan vi se at a1 og a2 må peke til det samme objektet.

### 6.3.1 Objekter som parametre

Når objekter blir brukt som parametre i metodekall, må en huske at det er referansen som blir sendt inn i metoden, og ikke en kopi av objektet. Dette vil si at hvis metoden endrer objektet, blir denne endringen også synlig utenfor metoden. Dette er forskjellig fra når en primitiv type blir brukt som eksempel. Da blir kopiert og sent inn til metoden. Endringer blir ikke endelige.

Eksempel:

```
import java.util.ArrayList;

class Klasse {

    public static void f(ArrayList a, int x) {
        a.add(new Node(14));
        x = 42;
    }

    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(new Node(13));
        int tall = 1;
        System.out.println(a1.size() + ", " + tall);
        f(a1, tall);
        System.out.println(a1.size() + ", " + tall);
    }
}
```

Dette programmet skriver ut:

```
1, 1
2, 1
```

## 7 Oppsummering

- Programmering kan ikke læres på forelesning
- Sett deg ned og programmer
- Prøv å gjøre det morsomt

Du kan finne noen oppgaver her:

<http://www.idi.ntnu.no/emner/tdt4120/arkiv/kode/trening.php>