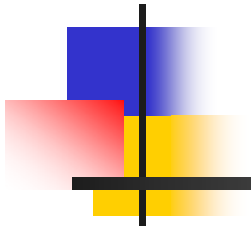


Uke 9 INF1000 – 13. okt 2009

Om separatorer I easyIO, Eliza (bruk av HashMap), + mer om metoder og klasser

Arne Maus

OMS, Ifi, UiO

1. Innlesning i easyIO, bruk av skilletegn

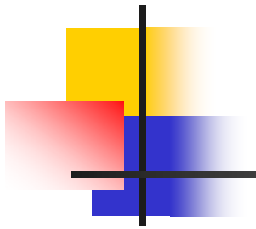
- Alle filer betraktes som en strøm av tegn (inludert de vi ikke alltid ser CR = vognretur, LF = linjeskift, og en serie ('ukjente') blanke tegn.)
- Lesing styres av en 'lese-pil'
  (som etter åpning av fila, peker på tegn nr. 1)
- Ved lesing, beveger lese-pilen seg høyre-over.
- Lese-pilen går aldri bakover.

1. Separatorer (skilletegn) i easyIO

1. Tegnene som leses av easyIO deles I to:
   A) De tegnene som skal skal leses og tolkes som data.
      - = de tegnene som ikke er separatortegn.

   B) Separator-tegnene som ligger foran og etter data (avgrenser det som leses som data).
      - Separatortegn er **alltid**: CR, LF og skjulte (rare) blanke tegn.
      - Når systemet starter er også vanlig blank (mellomrom) og tabulator separatortegn.
      - Hva som er separatotegn kan brukerne selv velge med unntak av de tegnene som alltid bukes.

2. Når en metode I easyIO kalles - eks. inInt(sep); så:
   1. Først leses det forbi evt. gamle (forrige) separator-tegn.
   2. Deretter leses det forbi evt. nye separator-tegn
   3. Nå står lese-pilen på første ikke-separator.
   4. Alle tegn fram til første nye separator-tegn leses som data.
   5. Lesepilen står etter lesing av data på første-ikke-separator funnet i pkt.4.



Lesing av ”hybeldata.txt” – systemets database

Først står det en linje for hver hybel, 21 linjer i alt, med hver linje på følgende form:

```
int etasje; char bokstav; String studentnavn; int saldo;
```

Et eksempel på en linje kan være:

```
2; C; Ole Brun; 2400;
```

For tomme hybler settes studentnavn lik "TOM HYBEL" og 0 i siste tall. Etter disse 21 linjene skal det stå en siste linje med format:

```
int måned; int år; int totaltAntallMåneder; int totalFortjeneste; 0;
```

Redusert ”hybeldata.txt” (legg merke til alle mellomrom (’blanke’) i fila) :

2; C; Ole Brun; 2400;
3; A; TOM HYBEL; 3000;

For å si det enklere:

Først hoppes det over først alle gamle og så alle nye skilletegn i ’sep’.
Deretter leses alt frem til neste nye skilletegn ***som de data vi egentlig er interessert i.***

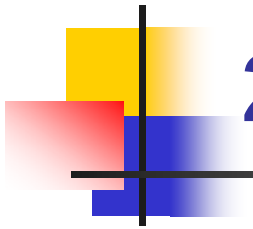
Eks. `int i =fil.inInt(” ;”);`

vil først hoppe over evt, gamle skilletegn og så det alle mellomrom og ; , så lese de neste tegnene fram til første ’blanke’ eller ; og prøve så å tolke det som står mellom disse to skilletegnene som et heltall.

```
M:\INF1000\Oblig3tips>java LesHybelData
Student:Ole Brun, med saldo:2400
Student:TOM HYBEL, med saldo:3000
```

```java
import easyIO.*;

class Student {
    String navn;
    int saldo;
    Student(String navn, int saldo) {
        this.navn = navn;
        this.saldo = saldo;
    }
    void skrivData() {
        System.out.println("Student:" + navn +", med saldo:" + saldo);
    }
}

// 2; C; Ole Brun; 2400;
// 3; A; TOM HYBEL; 3000;

public class LesHybelData {
    public static void main(String [] args) {
        int i = 0;
        int antLinjer = 2;


        In hdata= new In("hybeldata.txt");

        while (i++ < antLinjer ) {
            int et         = hdata.inInt("; ");
            char hbokstav  = hdata.inChar("; ");
            String navn    = hdata.inWord(";");
            int saldo      = hdata.inInt("; ");
            Student s = new Student(navn,saldo);
            s.skrivData();
        }

        hdata.close();
    }
}
```

Merk ingen blanke skilletegn når vi leser inn navnet - HVORFOR



2.Lage et psykolog-program

- Tar inndata – et spørsmål/en setning fra pasienten og svare som en dårlig psykolog
- Ha en fil med enkeltord, og hvis pasienten bruker et av ordene, så svare med tilhørende setning
- Bruke en HashMap for å holde par av ord, setning
- Eks:
  - far, Fortell meg om din far
- Må ha noe å si hvis vi ikke får treff:
  - Så interessant, fortell meg mer om dette.

> hei
Hei, jeg heter Sigmund, hva heter du
> Jeg heter Arne
Så det er altså navnet
> Ja
Fint, skal vi snakke om din barndom?
> Ja og særlig om mor
Fint, skal vi snakke om din barndom?
> Mor sier jeg jo
Lever fortsatt din mor
> nei
Så du mener ikke det
> hun slo meg ofte
Var det ofte at noen slo?
> ganske ofte
Interessant. Fortell mer.
>

Programskisse

```
import easyIO.*;
import java.util.*;
class Eliza {
  public static void main(String [] args) {

  }
}
class Samtale {
  HashMap hash = new HashMap();
  In tast = new In();
  void lesFraFil() {

  }
  void snakk() {

  }
}
```

```
import easyIO.*;
import java.util.*;

class Eliza {

  public static void main (String [] args) {
     if (args.length !=1) {
       System.out.println(" bruk: >java Eliza <fil-med-ord> ");
     } else {
       Samtale sam = new Samtale();
       sam.lesFraFil(args[0]);
       sam.snakk();
       }
  } // end main
}
```

```
class Samtale {
    HashMap <String,String>hash =
                new HashMap<String,String>();
    In tast = new In();

    void lesFraFil (String filnavn) {
      In fil = new In(filnavn);
        while (!fil.lastItem()) {
          String søkeord = fil.inWord();
          String svar = fil.inLine();
          hash.put(søkeord, svar);
        }
        fil.close();
        System.out.println
           ("Antall ord lest: " + hash.size());
    }
```

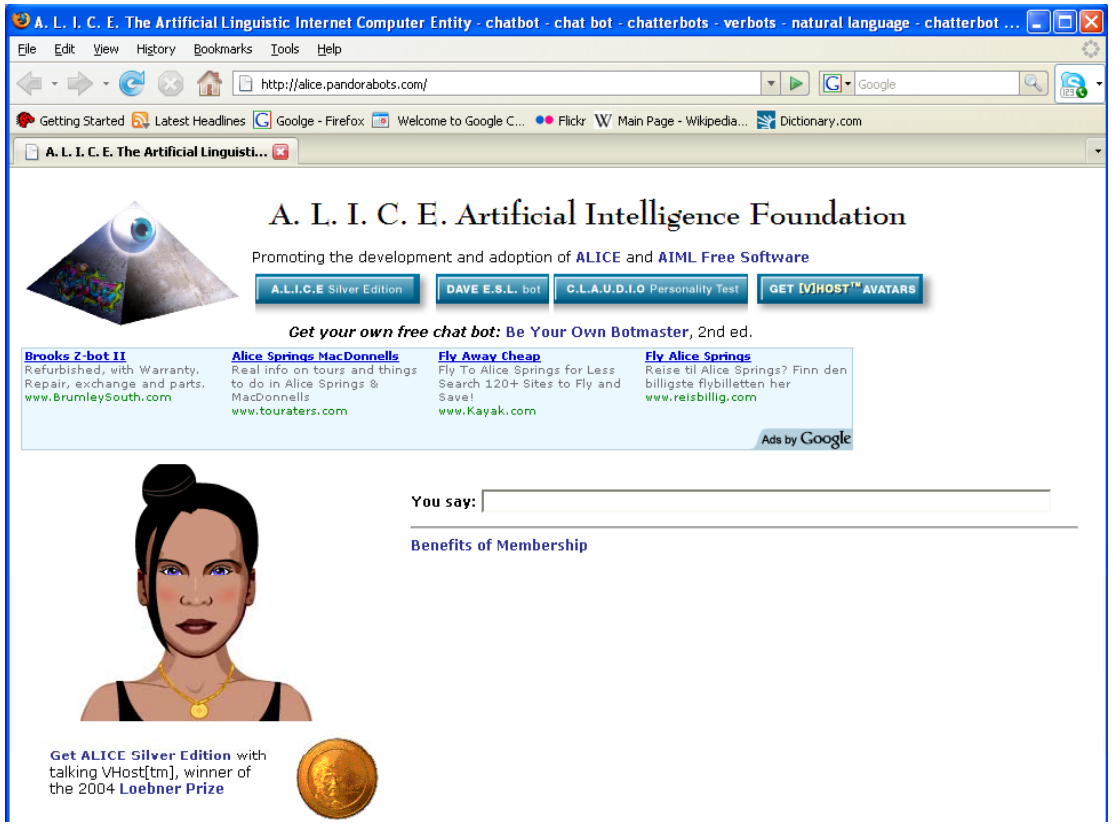
```
void snakk() {
        while (true) {
          System.out.print("> ");
          boolean funnetMatch = false;
          do {
             String ord = tast.inWord().toLowerCase();
             if (hash.containsKey(ord)) {
                 String svar = hash.get(ord);
                 System.out.println(svar);
                 funnetMatch = true;
             }
          } while (tast.hasNextChar() && !funnetMatch);

          if (!funnetMatch) {
             System.out.println("Interessant. Fortell mer.");
          }
          if (tast.hasNextChar()) {
             tast.readLine();  // Tømmer inputbufferet
          }
        }
    } // end snakk
}
```

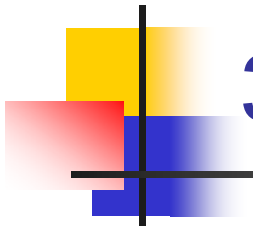
Ordfil.txt

```
far   Fortell meg mer om din far
faren Hadde du et vanskelig forhold til din far?
slo   Var det ofte at noen slo?
lei    Du sier du er lei deg, hvorfor det
mor    Lever fortsatt din mor
penger Er du bekymret for om du har nok penger
sint   Hvorfor bruker du 'sint' - er du selv sint
glad   Så bra
ikke   Forklar dette nærmere
vær    Blir du deprimer av dårlig vær
nei    Så du mener ikke det
ja     Fint, skal vi snakke om din barndom?
barn   Har du barn eller barnebarn?
barnebarn Hva heter de
datter  Hvor gammel er hun?
hei    Hei jeg heter Sigmund, hva heter du
jeg    Hvordan føler du deg
heter  Så det er altså navnet
gjenta Vil du heller snakke om din mor?
```

ALICE: En kunstig intelligens-basert prate-robot

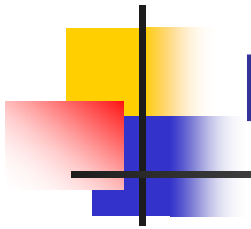
- ALICE = Artificial Linguistic Internet Computer Entity
- http://alice.pandorabots.com





3. Hva er en metode

- En metode er en valgfritt antall programsetninger vi gir et navn
- All kode i programmet er inne i en metode (som igjen er inne i en eller annen klasse)
- Skille mellom
  - å *deklarere* en metode (= skrive Javakode for og kompilere)
  - *Utføre* en metode (det som skjer når vi kaller den)
  - Når vi deklarerer en metode, skjer det 'ingen ting '
- En metode blir utført hver gang den kalles fra koden i en annen metode:
  - da hopper utførelsen av programmet til starten av den kalte metoden
  - har den kalt metoden parametere, kopieres verdiene brukt i kallet til metodens parameter-variable (de er som lokale variable i den kalte metoden)

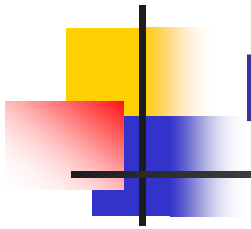


Hva skjer når vi kaller en metode

- Når vi kaller en metode, blir det opprettet et **metodeobjekt** og vi kopierer over verdiene brukt i kallet til parameterne
- Dette metodeobjektet
  - inneholder alle lokale variabler og parameterne til metoden
  - når setningene i metoden utføres, brukes disse variablene og parametrene av metoden
  - metodeobjektet fjernes automatisk når metoden er ferdig utført og returnerer
- Merk forskjellen på å deklarere en metode, og at den utføres.

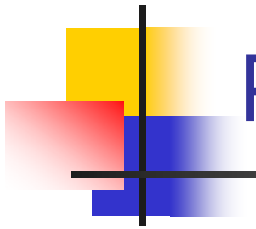
```
class C {

  int  skrivAntall(int i){
       System.out.println(" Du har kalt meg med:" + i);
       return i+10;
}}

class D
{
  static int dobbel( int k) {
       return 2*k;
  }

  void gjørMye(C cc, int v) {
       System.out.println(" gjørMye kalt");
       int j = cc.skrivAntall(v);
       System.out.println(" 1.verdien av j:" + j);
       j = dobbel(j);
       System.out.println(" 2.verdien av j:" + j);
       System.out.println(" 3.verdien av skrivAntall(j):"
                                  + cc.skrivAntall(j) );
  }

  public static void  main ( String[] args)  {
      C c = new C();
      D megSelv  = new D();
      megSelv.gjørMye(c,2);
}}
```

```
>java D
 gj°rMye kalt
 Du har kalt meg med:2
 1.verdien av j:12
 2.verdien av j:24
 Du har kalt meg med:24
 3.verdien av
   skrivAntall(j):34
```



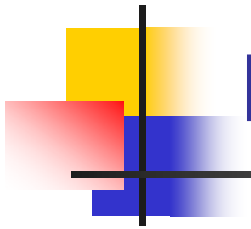
Hvorfor bruke metoder

- Vi deler opp programmet i metoder fordi:
  - Noen program setninger brukes *flere* steder, eller:
  - Vi vil dele opp programmet i mindre deler
    - Ingen metode bør være lenger enn 30 linjer (og helst mindre)
  - Hver del gjør noe veldefinert som fremgår av navnet:
    - regner ut en bestemt formel
    - skriver ut en meny
    - leser noen data fra terminal eller fil
    - tegner ut opplysninger på skjermen
    - .....



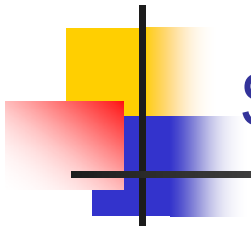
Problemløsning med metoder

- Når vi har laget en metode, og vi har forsikret oss om at den er 'riktig', så har vi laget en *ny operasjon*
- Vi kan nå i resten av koden tenke at vi nå har en slik operasjon tilgjengelig og nytte denne som om den var innebygd i Java
  - eks: skrive ut en meny, regne ut en bestemt formel,..
- Vi trenger da ikke tenke på alle detaljene om *hvordan* denne operasjonen blir utført, bare *at* den blir gjort.
- Vi har da laget et (lite) verktøy som kan gjenbrukes og lettere løse vårt større problem (hele systemet)
- Denne måte å programmere på heter *bottom-up* programmering og nyttes mye.
  - Eks: Java-biblioteket kan best forstås som en diger verktøykasse med nyttige operasjoner og datastrukturer vi kan (og ofte bør) bruke for å lage vårt program



Hva er en klasse

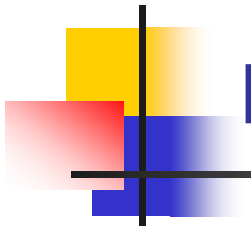
- En klasse er en beskrivelse av hvordan *ett* objekt av en bestemt type i vårt problem er.
  - Inneholder variable som beskriver egenskaper for ett slikt objekt – eks:
    - Navn, adresse, studiepoeng, kurs... for klassen Student
    - Registreringsnummer, eier, type, årsmodel for klassen Bil
  - Inneholder metoder som er fornuftig handlinger for ett slikt objekt – eks:
    - skrivUtVitnemål(), meldPåEmne(),.. i klassen Student
    - beregnÅrsavgift(), skiftEier(),.. i klassen Bil



Skille mellom deklarasjon og bruk av en klasse

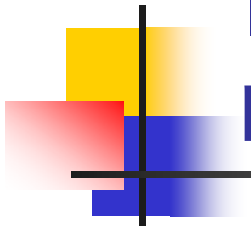
- Når vi deklarerer en klasse (= skriver Javakode for) skjer det 'ingen ting' i programmet
- Når vi oversetter og starter opp programmet vårt med javac og java, skjer 'lite':
  - De variable og metodene det står static foran er tilgjengelig
  - Ingen kode (med unntak av main) utføres
- Først når vi sier **new** på en klasse, får vi laget et objekt av klassen
  - Objektet inneholder alle variable og metoder som ikke har static foran deklarasjonen (objekt-variable og – metoder)
  - Når vi sier new, kaller vi en konstruktør-metode i klassen, og først når den er ferdig, returnerer new det med det nye objektet

```
class Konto1 {
  String eier;
  int kontoNum, saldo = 0;

  Konto1(String e) {
       eier = e;
  }

  void  settInn(int beløp) {
       saldo = saldo + beløp;
  }

  boolean taUt(int beløp) {
        // moderne bank med muligheter for overtrekk
        saldo = saldo - beløp;
        return saldo > 0;
  }
}

class Bank1
{
  Konto1 [] kontiene = new Konto1[100000];

  public static void  main( String[] args)  {
     Bank1 b = new Bank1();

     for (int i = 0; i < b.kontiene.length; i++) {
        b.kontiene[i] = new Konto1("kunde nr." + i);
        b.kontiene[i].settInn(100);
    }

 }
}
```



Forskjeller mellom klasser og metoder

- Begge lager objekter når de kalles, men:
- Et metode-objekt:
  - fjernes når metoden returnerer
  - inneholder ’bare’ variable og parametere som alle er skjult for resten av programmet
- Et objekt laget med **new** fra en klasse:
  - er i hukommelsen etter at det er laget (så lenge det minst er en peker som peker på det)
  - kan inneholde både metoder og variable, som kan nyttes av resten av programmet (med en peker og .)



Ikke alt i et objekt bør være synlig fra resten av programsystemet - innkapsling

- Vi ønsker ofte at resten av systemet bare skal se deler av et objekt
  - eks: **`int saldo`** i Konto1-objektet bør være skjult, resten av programmet skal bare bruke **`settInn() og taUt()`** metodene.
- Vi kan regulere tilgangen til variable og metoder ved å sette enten :
  - **`private`**
  - **`public`**
  - **`protected`**
- foran en metode eller deklarasjonen av en variabel

For ’små¨’ systemer hvor alle .java filene ligger på samme filområde, gjelder:

- Skriver vi:
  - **ingenting** foran en deklarasjon/metode, så er slike deklarasjoner fullt tilgjengelige for alle annen kode kompilert på samme filområde, men usynlig /sperret for kode kompilert på andre filområder.
  - **`private`** foran en deklarasjon/metode, så er den bare synlig fra kode i metoder deklarert i *samme klasse*, usynlig/sperret for all annen kode
  - **`protected`** foran en deklarasjon/metode, så er den synlig i samme klasser og subklasser og synlig i klassene på samme filområdet, men usynlig/sperret i andre klasser (på andre filområder).
  - **`public`** så er metoden/variabelen synlig for all annen kode.

- Slik delvis sperring av adgang til særlig variable, sikrer oss at vi kan bestemme fullt ut selv i en klasse hvordan en variabel skal endres.

Klassevariablenes levetid

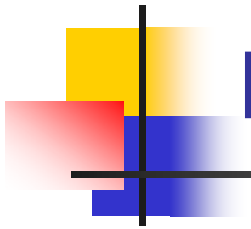
```
class Person {
  static int ant = 0;
  int alder;
  String navn;

  Person(){ ant ++; }
}
```

Denne variabelen blir deklarert når klassen Person blir referert til for første gang under kjøringen av programmet.

Variabelen lever helt til programmet avsluttes.

Første gang klassen Person blir referert til = første gang programeksekveringen "møter på" Person-klassen, f.eks. :

```
.... new Person() ....

.... i = Person.ant +  ....
```



Klassemetoder og objektmetoder

- Klassemetoder (static-metoder)
  - Definert selv om det ikke er laget noen objekter av klassen
  - Kan ”ses” av alle objekter av klassen
  - Kan brukes av andre gjennom dot-notasjon: <klassenavn>.metode(...)
  - Har ikke tilgang til objektvariable eller objektmetoder
- Objektmetoder
  - Bare definert i objekter av klassen
  - Kan ”ses” av objektet som metoden befinner seg i
  - Kan brukes av andre gjennom dot-notasjon: <peker>.metode(...)
  - Har tilgang til alle variable (både klassevariable og objektvariable) og alle metoder (både klassemetoder og objektmetoder)

Oppgave - Hva skriver programmet ut på skjermen?

```
class Studentregister {
  public static void main (String [] args) {
    Student s1 = new Student();
    s1.init("Torjus", "S25332");
    Student s2 = new Student();
    s2.init("Vilde", "S36336");s1.skrivUt();
    s2.skrivUt();
  }
}
class Student {
  static String navn;  static Stri
  static void init(String n, Strin
    navn = n;
    studId = s;
  }
  static void skrivUt() {
    System.out.println("Navn: " + navn);
    System.out.println("StudId: " + studId);
  }
}
```

```
$ javac Studentregister.java
$ java Studentregister
Navn: Vilde
StudId: S36336
Navn: Vilde
StudId: S36336
$
```

Å lage en fornuftig datamodell

- Med objekter kan vi ofte organisere våre data bedre.
- Eksempel:

```
String[] navn = new String[100];
String[]  fnr = new String[100];
int[] tlfnr = new int[100];
```

Informasjonen knyttet til en bestemt person er splittet opp i tre arrayer.

```
class Person {
  String navn;
  String fnr;
  int tlfnr;
}
Person [] personreg = new Person[100];
```

Informasjonen knyttet til en bestemt person er samlet i et objekt.

Bedre organisering – særlig når det er mye data å holde orden på.

Å lage en fornuftig datamodell (II)

- Med objekter kan vi samle data og operasjoner på dem.

```
... data om studenter...
... data om ansatte ...
... data om kurs ...
... student-metoder ...
... ansatt-metoder ...
... kurs-metoder ...
```

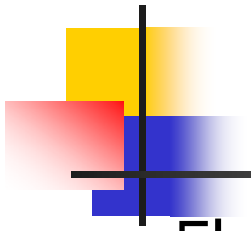
Her ligger alle data og alle metoder samme sted

```
class Student {
   ... data om studenter ...
   ... student-metoder ...
}
class Ansatt {
   ... data om ansatte ...
   ... ansatt-metoder ...
}
class Kurs {
   ... data om kurs ...
   ... kurs-metoder ...
}
```

Metoder og data som hører sammen er samlet.
Lett å se hvilke metoder som jobber på hvilke data (modularisering av koden).

Lett å kopiere alt som har med personer å gjøre (data + metoder) til andre programmer (gjenbruk).

Å lage en fornuftig datamodell (III)



- Eksempel: i Oblig 3 skal du holde orden på
  - en rekke studenter ⟶ **`class Student`**
  - en rekke hybler ⟶ **`class Hybel`**
  - et hybelhus (potensielt flere) ⟶ **`class Hybelhus`**

- En objektorientert løsning (med klassene over) sørger for at
  - variabler og metoder som logisk hører sammen ligger også samlet i programkoden
  - variabler og metoder som ikke har noe med hverandre å gjøre holdes godt atskilt i programkoden

Valg av datamodell: eksempel

- Eksempel:
- Du har gitt en fil med opplysninger om hvor mange registrerte tilfeller det var av tre ulike sykdommer i Norge hvert av årene 1950…2000:

	INFLUENSA	KYSSESYKE	MENINGITT
1950	...	...	...
1951	...	...	...
1952	...	...	...
.			
.			
2000	...	...	...

- Hvordan er det naturlig å modellere dette?

Noen muligheter

Forslag 1: Gruppere tellinger relatert til samme sykdom

```
class Sykdom {
    String sykdomsNavn;
    int[] antallTilfeller = new int[51];
}
```

Forslag 2: Gruppere tellinger foretatt samtidig

```
class Aarsdata {
    int antInfluensa;
    int antKyssesyke;int antMeningitt;
}
```

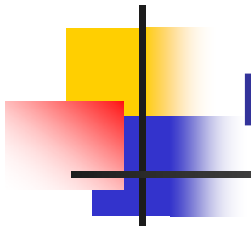
Forslag 3: Ingen gruppering – tre arrayer

```
int[] influensatilfeller = new int[51];
int[] kyssesyketilfeller = new int[51];
int[] meningittilfeller = new int[51];
```

Forslag 4: Ingen gruppering – en 2D-array

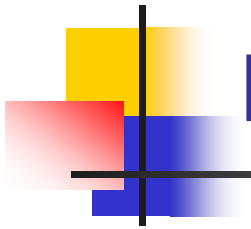
```
int[][] sykdomstilfeller = new int[3][51];
```

Beste datastruktur avhenger i stor grad av hva du skal bruke dataene til!



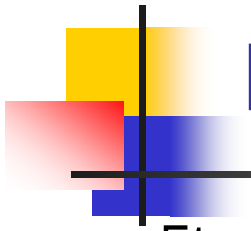
Råd 1: Skriv programmer "ovenfra og ned"

- Bestem først hvilke klasser som skal være med (og deres rolle) .
- Tegn UML-diagram!
- Fyll inn de mest sentrale variablene (de som utgjør datastrukturen), og skriv eventuelle nye klasser som trengs i datastrukturen
- Skriv metodene på toppnivå (dvs de som styrer den overordnede programflyten, f.eks. en kommandoløkke). Kall på metoder ved behov, selv om disse ennå ikke er skrevet.
- Skriv metodene du kaller på ovenfor, og fortsett til programmet er ferdig.



Råd 2: skriv metoder "utenfra og inn"

- Når du skal skrive en metode, bestem først av alt hva som er input og output til metoden:
  - Input:
    - Eventuelle parametere til metoden
    - Kan også være klassevariable/objektvariable
  - Output:
    - Eventuell returverdi fra metoden
    - Kan også være modifikasjoner av klassevariable/ objektvariable (f.eks. endring av innholdet i en HashMap).



Råd 3: Deleger oppgaver

- Et viktig kjennetegn ved god programmering er at man delegerer oppgaver når det er naturlig – dvs kaller på metoder for å utføre deloppgaver.
- Dermed blir hver enkelt del av programmet oversiktlig, og faren for feil minimeres. Det blir også lettere å finne feil senere.
- Eksempel:
  - Hvert case i en kommandoløkke kaller på en metode som utfører den ønskede kommandoen, i stedet for at alt gjøres inni selve kommandoløkken.

- NB: ikke overdriv delegering. Det er f.eks. ofte ikke naturlig at hvert eneste objekt har metoder for å lese fra terminal – det kan i mange tilfeller være bedre å gjøre slike ting sentralt (og heller kalle på metoder i objektene for å oppdatere deres variable).

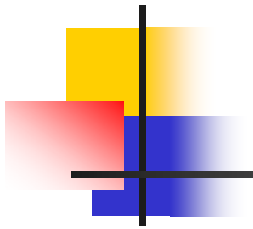
Råd 4: formater alltid koden underveis

Dårlig

```
    class Eksempel {
public static void main (String [] args) {
    int x = 0;
  for (int i=0; i<10; i++) {
x = x + 1;
      } if (x < 0)
  {System.out.println("Det var rart");
   }}}
```

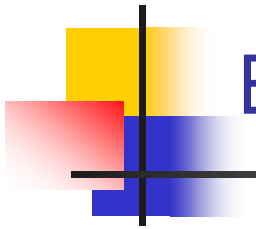
Bra

```
class Eksempel {
  public static void main (String [] args) {
    int x = 0;
    for (int i=0; i<10; i++) {
      x = x + 1;
    }
    if (x < 0){
      System.out.println("Det var rart");
    }
  }
}
```



Råd 5: Det er alltid lov å gå tilbake å endre på noe!

- Programmer blir til ved at vi jobber litt her og der.
- Vi finner ofte ut at vi trenger flere klasser, eller at en klasse bare er “i veien” og fjerner den
- Det er ingen skam å snu. Det endelige programmet kan ha andre klasser og metoder enn vi startet med
- Pass likevel på å holde programmet kompilerbart og å heller ha ”tomme skall” av alle metoder som kalles enn å ikke ha de der.



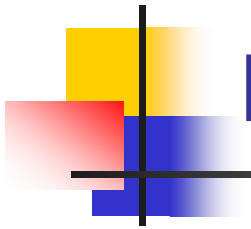
Eksempel: Flyreservasjon

Klasser

Egenskaper

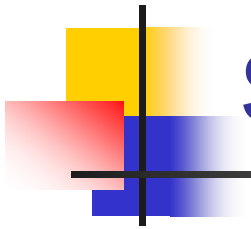
Metoder

- Vi skal lage et system for et flyselskap
- Systemet skal holde orden på alle selskapets flyvninger og reserverte seter på flyene
- En flyvning har en kode, et avreisested og en destinasjon, i tillegg til et fly, som har et identifikasjonsnummer
- Et fly består av seterader, med seter
- Oppgavene systemet skal løse er å lese inn en beskrivelse av alle flyene, med antall seter, klasser på de forskjellig seteradene, osv
- Så skal man kunne reservere seter, avbestille og skrive ut en oversikt over flyets seter, med klasse og om det er ledig eller ikke



Eksempel: Flyreservasjon

- class Systemet
  - Inneholder kun main-metoden. Lager objekt av klassen under og kaller på ordreløkke-metode.
- class Flyreservasjon
  - Inneholder ordreløkke og andre metoder + HashMap-tabeller for å holde orden på flyvningene.
- class Fly
  - Hvert objekt inneholder info om en flyet + alle seteradene og setene i flyet
- class Seterad
  - Setene i raden
- class Sete
  - Klasse og om det er opptatt eller ikke



Systemet

```
import easyIO.*;
import java.util.*;

class Systemet {
  public static void main (String[] args) {
    String s1 = ”Fly.txt";
    String s2 = ”Bestillinger.txt";
    Flyreservasjon f = new Flyreservasjon(s1,s2);
    f.ordreløkke();
  }
}
```

Flyreservasjon

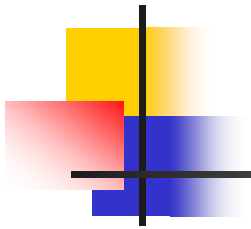
```
class Flyreservasjon {
  HashMap fly = new HashMap();
  HashMap flyvninger = new HashMap();

  Flyreservasjon(String s1, String s2)
    lesFly(s1);
    lesReservasjoner(s2);
  }
  void lesFly(String fnavn) {...}
  void lesReservasjoner(String fnavn) {...}
  void ordreløkke() {...}

  ...
}
```

Datastruktur

Konstruktør som gjør initialisering (her: lese data fra fil)

Metoder for å lese fra fil og for å lese inn kommando fra bruker

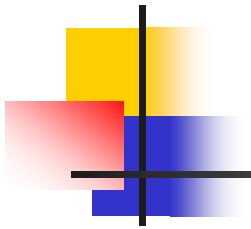
Her kommer det metoder som skal kalles fra ordreløkken



Flyreservasjon

- Programmere ordreløkken
  - For hver kommando som skal utføres, skal ordreløkken kalle på en passende metode i klassen Flyreservasjon.
  - For at programmet skal kompilere, sørg for å deklarere alle de metodene som du kaller på fra ordreløkke-metoden. Du kan vente med å fylle inn innholdet i disse metodene, dvs bare fyll inn en utskriftssetning i hver av metodene.
  - Eksempel: hvis ordreløkken kaller på metoden visFlyvning(), så deklarerer du samtidig denne ”dummy-metoden” i klassen Flyreservasjon:

```
void visFlyvning() {
  System.out.println(”Metoden visFlyvning utført”);
}
```

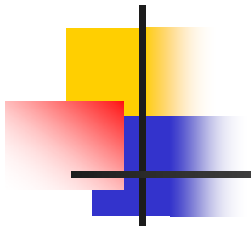


Skrive ut flyvning

- Programmer metodene som kalles fra ordreløkken
- Eksempel (i klassen Flyreservasjon):

```
void visFlyvning() {
  System.out.println(”Flyvning: ”);
  String flightKode = tast.inLine();

  Flyvning flight = <finn flyvingen ved oppslag i
                flyvninger>;

  flight.skrivUt();
}
```

Oppdraget delegeres videre til en metode i Flyvning-objektet som er aktuelt.

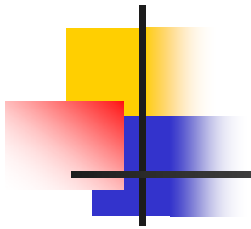


Flyvning

- Skriver ut litt informasjon om flyvningen og delegerer så ansvaret for utskrift av oppsettet i flyet til klasen fly.

```
class Flyvning {
  String flightkode;
  String avreisested;
  String destinasjon;
  Fly fly;
  void skrivUt() {
    System.out.println("Flight: " + flightkode);
    System.out.println("Fra: " + avreisested);
    System.out.println("Til: " + destinasjon);
    fly.skrivUt();
  }
}
```

Oppdraget delegeres videre til en metode i Fly-objektet som er aktuelt.

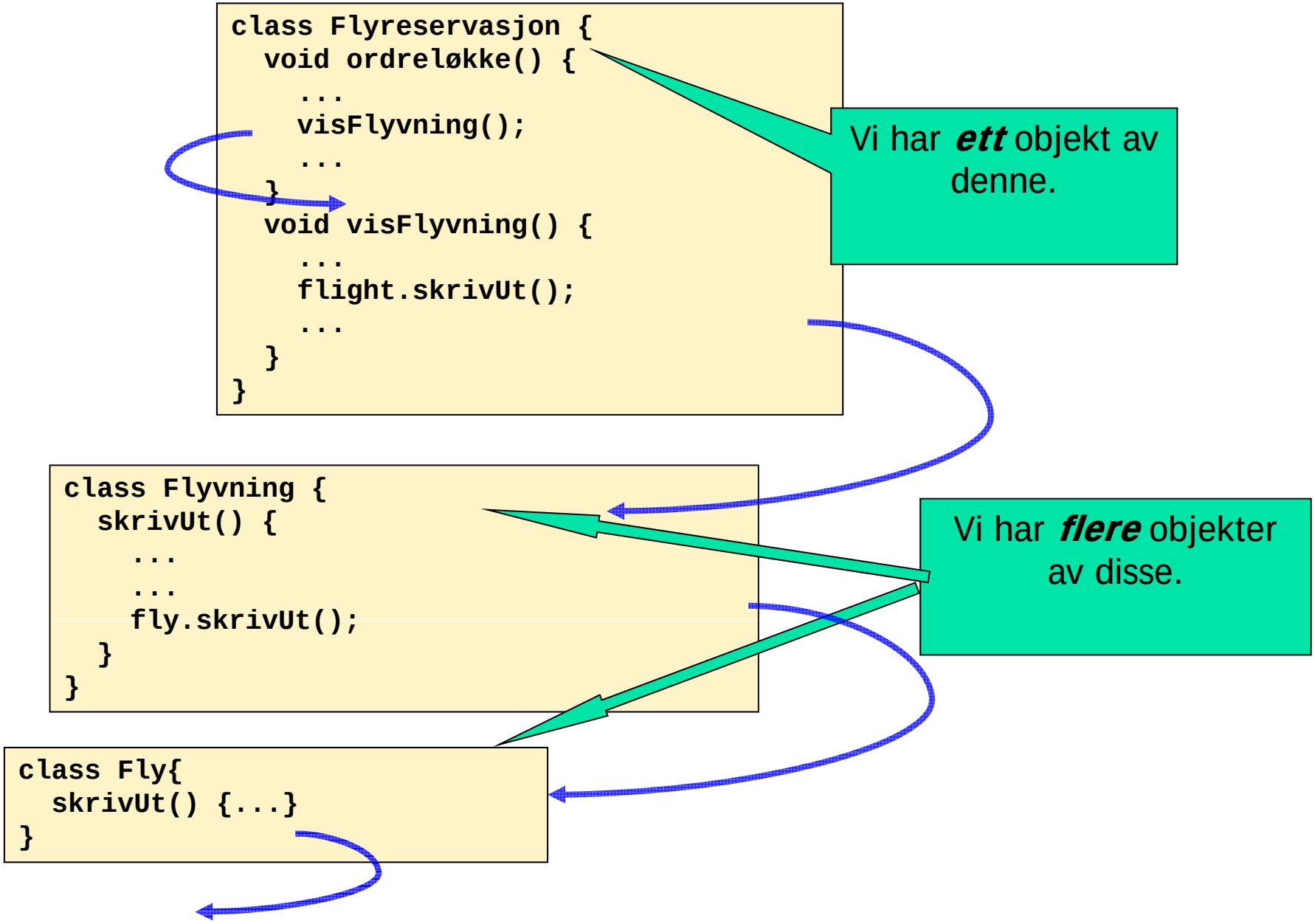


Fly

- Skriver ut informsjon on flyet og delegerer videre til seteradene, som igjen delegerer videre til setene.

```
class Fly {
  String flykode;
  Seterad[] seterader;
  int skrivUt() {
    System.out.println("Flykode: " + flykode);
    for(int i=0; i<seterader.length; i++){
      seterader[i].skrivUt();
    }
  }
}
```

Og Fly delergerer videre


class Flyreservasjon {
void ordreløkke() {
...
visFlyvning();
...
}
void visFlyvning() {
...
flight.skrivUt();
...
}
}
Vi har ***ett*** objekt av denne.
class Flyvning {
skrivUt() {
...
...
fly.skrivUt();
}
}
Vi har ***flere*** objekter av disse.
class Fly{
skrivUt() {...}
}