INF1000 – Uke 10

Litt mer om innkapsling, klassevariable og -metoder

Hvordan gripe an et stort problem? 5 gode råd

Et større programeksempel: Administrere hoppkonkurranse

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 et Konto-objekt bør være skjult, resten av programmet skal bare bruke metodene **`settInn()`** og **`taUt()`**.
- Vi kan regulere tilgangen til variable og metoder ved å sette en av følgende foran en variabel- eller metodedeklarasjon:
  - **`private`**
  - **`public`**
  - **`protected`**

2

For "små" systemer hvor alle java-filene ligger på samme filområde, gjelder:

- Skriver vi:
  - **ingenting** foran en deklarasjon, så er deklarasjonen fullt tilgjengelige for alle klasser i den samme pakken/katalogen, men utilgjengelig for klasser i andre pakker/kataloger.
  - **`private`** foran en deklarasjon, 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 fra klasser i samme pakke/katalog, men utilgjengelig fra klasser i andre pakker/kataloger (med mindre det er en subklasse).
  - **`public`** så er deklarasjonen synlig for all annen kode.
- Slik delvis sperring av adgang til variable/metoder, sikrer oss at vi kan beskytte data for tilgang utenfra.

3

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 +  ....
```

4

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)

5

Råd 1: Programmer ”ovenfra ned”

- Hvilke klasser skal være med?
  - Les oppgaven
  - Se etter substantiver
  - Lag klassediagram
- Bestem datastruktur
  - Hvordan er input-dataene?
  - Fyll inn de mest sentrale variablene
  - Trengs nye klasser?
- Følg programflyten når du bestemmer metoder
  - Skriv først metodene på ”toppnivå” f.eks. en kommandoløkke
  - Kall på metoder ved behov, selv om disse ennå ikke er skrevet
  - Skriv metodene du kaller på, og fortsett til programmet er ferdig

Velg datastruktur etter hva som skal gjøres!

- I objekt-orientert programmering
  - *tenker vi i form av objekter*
  - *men programmer i form av klasser*
- Spør: Hva kan objektene gjøre for meg?
- Prøv å gruppere data etter objekter som “eier” dem
  - variable og metoder som logisk hører sammen bør ligge samlet
  - variable og metoder som ikke har noe med hverandre å gjøre bør holdes godt atskilt

Gruppering etter ”eier”

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

Ikke objektorientert:
Info om person splittet opp i tre arrayer

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

Info om person samlet i samme objekt

Data og metoder hører sammen

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

↓

```
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 bør samles
- Lett å se hvilke metoder som jobber på hvilke data
- Lett å kopiere alt som har med personer å gjøre (data + metoder) til andre programmer

Valg av datamodell: nytt eksempel

Gitt fil med opplysninger om antall registrerte tilfeller det var av tre ulike sykdommer i Norge :

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

- Hvordan er det naturlig å modellere dette?
- Beste datastruktur avhenger av hva du skal bruke dataene til!

Gruppere tellinger relatert til samme sykdom

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

Gruppere tellinger foretatt samtidig

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

Ingen gruppering – tre arrayer

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

Ingen gruppering – en 2D-array

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

Råd 2: Metoder "utenfra og inn"

- Hva er input og output til metoden du skal skrive?
- 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

- Stykk opp oppgavene og fordel dem
- Dermed blir hver enkelt del mer oversiktlig
  - faren for feil minker
  - lettere å finne feil senere
- Ofte lurt: Deleger operasjoner på data til objekter som er ”nærme” dataene
- Ikke overdriv delegering!
  - hvert objekt trenger ikke metoder for å lese fra terminal!
  - av og til bedre å gjøre ting sentralt og kalle på metoder i objektene for å oppdatere deres variable

Ideen bak objektorientering

- Hvert objekt skal ha sin bestemte og naturlige oppgave
- Kollektivt samarbeid om å løse oppgavene
- Objektene opprettes som instanser av klasser
- Objektene samarbeider ved å sende meldinger:
  - kaller hverandres metoder
  - overfører informasjon i form av parametere og returverdier
- Metodekallene har en entydig mottager som overtar ansvaret for oppgaven

Helhet og deloppgaver i OOP

- *Vi trenger ikke å ha oversikt over hele programmet eller hele datastrukturen* når vi skriver en metode
- Vi ”skifter hatt” og ser systemet gjennom øynene til hver av aktørene for seg
  - Når vi er kelner, beskriver vi kelneren ut fra kelnerens perspektiv, når vi er hovmesteren ser vi det ut fra hans perspektiv osv.
- Vi programmerer en klasse ut fra klassens perspektiv og glemmer da resten av helheten

Objekter svarer til programmer

- Vi kan tenke oss at hvert objekt av en gitt klasse svarer til en ”kjøring av klassen”
- Et objekt kan *samarbeide* med alle det kjenner til ved metodekall
- Vi starter ett av ”programmene” ved å kalle klassens main-metode
- Deretter: objektene *opprettes* når programmet kjører

Råd 4: Formater koden

```
    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");
   }}}
```

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!

Råd 5: Ingen skam å snu!

- Programmer blir til ved at vi jobber litt her og der
  - Vi kan bruke mange runder før vi er fornøyd
- Vi finner ofte ut at vi trenger flere klasser
  - eller at en klasse bare er "i veien" og fjerner den
- Det endelige programmet kan ha andre klasser og metoder enn vi startet med
- Pass likevel på å holde programmet kompilerbart!
  - Lag tomme metoder som du kan fylle ut siden
  - Hold koden ryddig

Flyreservasjon

Klasser

Egenskaper

Prosedyrer

- 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
- Systemet skal lese inn beskrivelse av flyene, med antall seter, klasser på de forskjellig seteradene, osv
- Det skal kunne reservere seter, avbestille og skrive ut en oversikt over flyets seter, med klasse og om det er ledig eller ikke

Klasseinndeling

- 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
- class Seterad
  - Setene i raden
- class Sete
  - Klasse og om det er opptatt eller ikke

class 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();
  }
}
```

class Flyreservasjon

```
class Flyreservasjon {
  HashMap fly = new HashMap();
  HashMap<String, Flyvning> flyvninger
    = new HashMap<String, Flyvning>();

  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: ordreløkken

- For hver kommando skal ordreløkken kalle på en metode i klassen Flyreservasjon.
- 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
- Eksempel: kaller ordreløkken på metoden visFlyvning(), kan du skrive en ”dummy-metode”:

```
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.

class 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();
  }
}
```

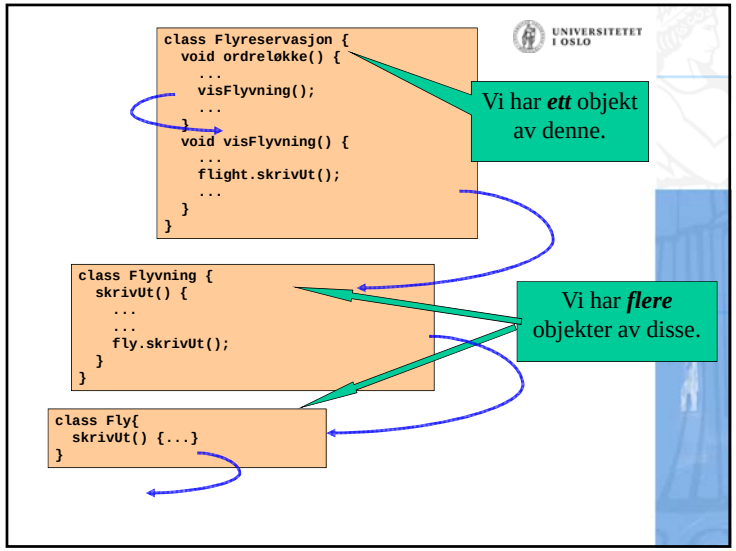
Oppdraget delegeres videre til en metode i Fly-objektet

class Fly

- Skriver ut informsjon on flyet
- Delegerer videre til seteradene
- 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



Administasjon av hopprenn

Konkret eksempel på OOP-program. Programmet skal

- registrere navn og idrettslag til skihoppere
- trekke startlisten til første omgang
- lese inn lengde og 5 stilkarakterer for hvert hopp
- beregne poengsum og skrive ut resultatlisten
- beregne startrekkefølgen i 2. omgang ved å snu resultatlisten fra 1. omgang
- kunne simulere hopprennet ut fra tilfeldige tall

Metoden for beregning av poengsum ut fra lengde og stilkarakterer er beskrevet i oppgave 5.9 i læreboka.

28

Hva er objektene?

- I et generelt tilfelle vil objektene i en OOP-modell motsvare både konkrete og abstrakte ting i verden
- I et hopprenn er det mange *skihoppere*, disse er de konkrete objektene.
- Vi har også en rekke *hopp* som, om de ikke er helt konkrete, i alle fall er noe vi kan referere entydig til.
- Litt mer abstrakt består hopprennet av to *omganger*.
- Vi trenger også å kunne henvise til selve *konkurransen*

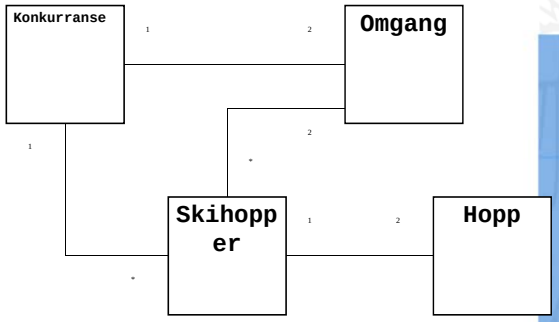
29

Substantivmetoden

- En god tommelfingerregel er å merke seg substantivene, for disse gir ofte opphav til en god klasse-inndeling.
- Klasseinndeling bør samsvare med begrepene som brukes i en beskrivelse av problemet i naturlig språk.
- Vi bør alltid angripe en slik oppgave med å identifisere klassene og tegne forholdet mellom dem i et enkelt klassediagram

30

Klassediagram av hopprenn-systemet



31

class Hopprenn og main-metoden

```
class Hopprenn {
    public static void main( String[] args ){
        Konkurranse rennadm = new Konkurranse();
        rennadm.kommandoløkke();
    }
}
```

- Det opprettes et objekt av klassen Konkurranse som heter rennadm. Dette skjer idet setningen new Konkurranse() utføres.
- Når objektet rennadm opprettes, utføres en bestemt metode i class Konkurranse som kalles *konstruktøren*.
- Vi kaller metoden til objektet rennadm. Merk at metoden har en adresse (nemlig rennadm) og at denne er ansvarlig for at kommandoene utføres.
- Metoden kommandoløkke() ligger i class Konkurranse.

32

Konstruktøren

- Konstruktøren heter alltid det samme som klassen
- Den skrives uten typeangivelse. Konstruktøren i Konkurranse-klassen heter kun Konkurranse()
- Den utføres en og bare en gang for hvert objekt (nemlig når det opprettes)
- Kan utelates fra klassen
- Brukes normalt til initialiseringer

33

Konkurranse-klassen: Hovedmeny

I hovedmenyen skal vi foreta registrering av nye skihoppere og kunne starte omganger:

```
*** MENY ***
0. Avslutt
1. Registrer ny deltager
2. Trekning av startnummer
3. List alle deltagere
4. Første omgang
5. Andre omgang
6. Generer fiktive deltagere
```

34

Skisse til klassen

```
import easyIO.*;
class Konkurranse {

    Konkurranse() {
        System.out.println("HOPP-PROGRAM VERSJON 1.0");
    }
    void kommandoløkke(){
        …
    }
    // Her kommer alle de andre metodene i class Konkurranse
}
```

35

```
    do {
          System.out.print("\nValg (9 for Meny): ");
          valg = tast.inInt();
          switch(valg){
              case 0: System.out.println("Programmet
avslutter");
                      System.out.println(); break;
              case 1: registrerDeltager(); break;
              case 2: trekning(); break;
              case 3: listDeltagere(); break;
              case 4: /* kode følger siden */ break;
              case 5: /* kode følger siden */ break;
              case 6: autogenerer(); break;
              case 9: skrivMeny(); break;
              default: System.out.println("Du tastet feil");
          }
      } while (!(valg == 0));
```

36

Grunnleggende datastruktur

Vi må ha en datastruktur som effektivt tillater å
- legge inn data om skihoppere
- assosiere skihoppere med hopp
- assosiere startlister og resultatlister med skihoppere (men ikke nødvendigvis motsatt)

➔ Vi trenger å ha lett tilgang til informasjonen om hopperne
➔ Vi trenger å gruppere hoppere i hver omgang

Den enkleste løsningen er å gruppere skihoppere i arrayer:
- arrayene administreres innen hver omgang
- vi utnytter array-ordningen i startlistene
- vi kan enkelt lage nye ordninger av hoppere fra gamle, for eksempel å snu resultatlisten fra 1. omgang og la den være startlisten i 2. omgang

37

Datastruktur i class Konkurranse

```
class Konkurranse {
    final int MAX_ANTALL = 60;
    Skihopper[] deltager = new
  Skihopper[MAX_ANTALL];
    int antallHoppere = 0; // Brukes som indeks i
  deltager     …

  void kommandoløkke() {…}

     void registrerDeltager(){
        deltager[antallHoppere++] = new
  Skihopper();
    }
    …
}
```

Delegering av ansvar: Skihopper-objektet er selv ansvarlig for å innhente opplysinger om seg selv fra brukeren!

38

Konstruktøren i Skihopper-klassen henter info

```
class Skihopper {
    private static final int UDEFINERT = -1;
    String navn,idrettslag;
    int startnr = UDEFINERT;

    Skihopper(){
        In tast = new In();
        System.out.println("*** NY DELTAGER ***");
        System.out.print("  Navn: ");
        navn = tast.inLine();
        System.out.print("  Klubb: ");
        idrettslag = tast.inLine();
    }
```

39

Registrering av ny deltager

Deltagere skal registreres med navn og klubb. Det skal foretas en trekning ut fra tilfeldig genererte tall, hvoretter deltagere gis et startnummer.

```
Valg (9 for Meny): 1
*** NY DELTAGER ***
  Navn: Geir Ellingsrud
  Klubb: Matematisk institutt

Valg (9 for Meny): 1
*** NY DELTAGER ***
  Navn: Arild Waaler
  Klubb: Ifi
```

40

case 3: listDeltagere() og metoden toString()

```
    void listDeltagere(){
       for(int i=0; i<antallHoppere; i++)
           System.out.println( deltager[i] );
    }

class Skihopper {
  …
  public String toString(){
        String s = "";
        if( startnr != UDEFINERT ) s += startnr + " ";
        s += navn +" "+ idrettslag;
        return s;
    }
}
```

Her skriver vi ut et Skihopper-objekt direkte i utskriftssetningen.
Dette krever at vi har skrevet en metode i class Skihopper med signaturen:
public String toString()
Java-systemet vil automatisk utføre denne metoden når objektet skal skrives ut!

case 2: trekning ();

Her opprettes en full array av alle deltagere. Dette gjøres for å slippe å overføre parameteren antallHoppere (en forenkling vi kan ønske å endre på senere ved utvidelse av programmet!)
NB! Vi slipper dette hvis vi bruker Arraylist isteden!

```
void trekning(){
      Skihopper[] temp = new Skihopper[antallHoppere];
      for( int i=0; i<antallHoppere; i++ )
          temp[i] = deltager[i];
      deltager = temp;
      stokk();
      for( int i=0; i<antallHoppere; i++ )
          deltager[i].startnr = i+1;
      System.out.println("Trekning ferdig (det kan ikke registreres
            nye deltagere) ");
      trukket = true;
  }
```

Vi oppdaterer så startnumre i Skihopper-objektene (bør ideelt gjøres med via en "set-metode"!)

void stokk() og Random tall-generator

Random-klassen har en rekke funksjoner for å generere tilfeldige data på ulike format.
nextInt(int max) gir en int k i intervallet 0<= k < max

```
import java.util.Random;
Random tall = new Random();

 void stokk( ) {
      int j,k;
      Skihopper temp;
      for( int i=0; i<100; i++ ){
          j = tall.nextInt(deltager.length);
          k = tall.nextInt(deltager.length);
          temp=deltager[j];
       deltager[j]=deltager[k];
       deltager[k]=temp;
      }
   }
```


Tall-generatoren kan brukes til å lage testdata

```
String[] fornavn = { "Odin", "Alf", "Even", "Ulf", "Elg", "Tor",
   "Rolf" };
String[] suffiks = { "snes", "sen", "svik", "shaug", "sdal",
   "sbakken", "sli", "sletten" };
String[] klubb = { "TIL", "HIL", "FIL", "BIL", "MIL", "KIL" };


Skihopper genererNyHopper(){
   String navn = fornavn[tall.nextInt(fornavn.length)] +" "+
                      fornavn[tall.nextInt(fornavn.length)] +
                       suffiks[tall.nextInt(suffiks.length)];
   String idrettslag = klubb[tall.nextInt(klubb.length)];
   return new Skihopper( navn, idrettslag );
}
```

Vi lager en ny konstruktør for class Skihopper som kan ta inn data utenfra.

Opprettelse av Omgang-objekter

```
case 4: if( !trukket ) break;
         if( førsteOmgang == null ) førsteOmgang = new Omgang(
   deltager, true );
           førsteOmgang.kommandoløkke(); break;
case 5: if( førsteOmgang == null ) break;
         if( andreOmgang == null )
        andreOmgang = new Omgang(
   reverser(førsteOmgang.rekkeflg), false );
         andreOmgang.kommandoløkke(); break;

class Omgang {
…
   Omgang( Skihopper[] startliste, boolean førsteomgang){
       // initialisering
}
```


Omgang-klassen: Konstruktør

```
class Omgang {
    Skihopper[] startliste;
    Skihopper[] rekkeflg;
    int antall;  // antall som har hoppet
    boolean førsteomgang;   // overføres til konstruktøren

  Omgang( Skihopper[] startliste, boolean førsteomgang){
       this.startliste = startliste;
       this.førsteomgang = førsteomgang;
       rekkeflg = new Skihopper[startliste.length];
    }
```

Merk at vi via startlisten får tilgang til alle skihopperne som skal starte i denne omgangen. Vi overfører altså hele datastrukturen med Skihopper-objekter.

Meny for hver omgang

I hver omgang skal vi foreta registrering av nye hopp, holde orden på hvem som er neste hopper, samt resultatlisten.

*** MENY 1. OMGANG ***
0. Tilbake til hovedmenyen
1. Registrer nytt hopp
2. List gjenstående hoppere
3. Resultatliste
4. Simuler resten av omgangen

Hoppene skal registreres med lengde og 5 stilkarakterer. Startlisten for 2. omgang lages ved å snu resultatlisten for 1. omgang opp ned.

class Skihopper og class Hopp

```
class Skihopper {
   String navn,idrettslag;
   int startnr;
   Hopp førstehopp, andrehopp;
   …
}

class Hopp {
    double lengde;
    double[] karakter;
    double poeng;
   …
}
```

startnr må angis etter at objektet er opprettet
Delegering av ansvar tilsier at vi bør legge rutinen for utskrift av data om skihopperen til skihopperen selv (eller informasjonen som skal skrives ut)
Vi må støtte separat utskrift fra både 1. og 2. omgang. Dette kan vi oppnå enten ved å overføre en parameter som sier hvilken omgang vi ønsker utskrift for eller ved separate metoder som kalles fra hver omgang.

Kommandoløkken i Omgang-objektet

```
skrivMeny();
      do {
          System.out.print("\nValg (9 for meny): ");
          valg = tast.inInt();
          switch(valg){
              case 0: System.out.println(); break;
              case 1: nesteHopp(); break;
              case 2: skrivGjenstående(); break;
              case 3: skrivResultat(); break;
              case 4: simulerOmgang(); break;
              case 9: skrivMeny(); break;
              default: System.out.println("Du tastet feil");
          }
      } while (!(valg == 0));
  }
```


Registrering av nytt hopp

```
void nesteHopp(){
      if( antall >= startliste.length ) return;
      startliste[antall].nyttHopp(førsteomgang);
      oppdaterListe(); // sett sortert inn i resultatlisten
  }
```

- Innlesning av data om hoppet og beregning av poengsum delegeres til Skihopperen og derfra videre til Hopp-klassen
- Når vi registrerer et nytt hopp, øker vi en lokal tellevariabel ”antall” med én og setter en referanse til den aktive hopperen inn i en array ”rekkeflg” slik at den holdes sortert på hoppernes poengsum.
- For å sikre at Skihopper-objektet returnerer riktig poengsum, overføres den boolske variabelen ”førsteomgang”.

Innlesing av hopp-data i Hopp-klassen

```
Hopp() {
       In tast = new In();
       System.out.print("  Lengde: ");
       lengde = tast.inDouble();
       karakter = new double[5];
       for(int i=0; i<5; i++){
           System.out.print("  Dommer " + (i+1) + ": ");
           karakter[i] = tast.inDouble();
       }
       poeng = Poengberegning.poengsum(lengde, karakter);
   }
```


Poengberegning kan legges i en egen klasse

```
class Poengberegning {

    private static final int TABELLPUNKT = 120;
    private static final double FAKTOR = 1.8;
    private static final int STARTPOENG = 60;

  static double poengsum( double lengde, double[] karakterer ){
       double tillegg = (lengde - TABELLPUNKT)*FAKTOR;
       double lengdepoeng = STARTPOENG + tillegg;
       double sum = lengdepoeng + stilsum( karakterer );
       return sum;
    }
…
}
```

Begegning av stilkarakterer: kutt laveste og høyeste stilkarakter og summér

```
private static double stilsum( double[] karakterer ){
       int ant = karakterer.length;
       double[] sortert = new double[ant];
       for (int i=0; i<ant; i++){
           sortert[i] = karakterer[i];
           int j=i;
           while ( j>0  ) {
               if (sortert[j]<sortert[j-1]){
                   double temp = sortert[j-1]; sortert[j-1] =
 sortert[j]; sortert[j] = temp;}
               j--;
           }
       }
       double sum = 0;
       for (int i=1; i<ant-1; i++) sum = sum + sortert[i];
       return sum;
   }
```

53

Oppsummering

- I hopprennet utspenner vi også en liten verden, en gruppe av objekter med dedikerte oppgaver.
- Vi fant klasseinndelingen til hopp-programmet fra substantivene i oppgaveteksten.
- Når vi gikk gjennom programmet gikk vi gjennom klassene ”ovenfra-ned” og skisserte en klasse først når vi fikk behov for et objekt av den.
- Vi definerte navn på metoder før vi visste hvordan de kunne kodes, og så på koden i rekkefølgen ”utenfra-inn”: vi sporet koden i omtrent samme rekkefølge som programmet eksekverer.
- Når dere selv skal skrive programmer er det ofte lurt å skrive kode i samme rekkefølge! Dette gir dere en metode dere kan følge som tar dere fra oppgavetekst til program i små, naturlige steg.

54

Å tenke objekt-orientert

- Vi tenker objekt-orientert i valg av datastruktur når vi lar objektenes funksjon være det vi primært har for øye.
- Vi må hele tiden spørre: hva kan dette objektet gjøre for meg?
- Når vi velger datastruktur, så tenk på at alle data også kan betraktes som objekter. Hvilke tjenester tilbyr de, hva kan de gjøre for meg?
- Dette passer riktignok ikke perfekt inn i enhver situasjon, snarere er det pedagogiske tommelfingerregler, men de kan likevel hjelpe deg til å gjøre gode valg av datastruktur og klasser underveis.

55

Noen andre funksjoner vi kan implementere

- Hvordan har en bestemt hopper gjort det i hver omgang? Deler av oppgaven kan delegeres til Skihopper-objektet og Omgang-objektet
- Hvordan har de ulike dommerne dømt? Vi må scanne gjennom alle skihopperne og finne alle hopp
- Utvidelse av programmet til å administrere flere ulike årsklasser (for eksempel ”Gutter 14. år”): Opprett et nytt Konkurranse-objekt for hver årsklasse
- Tilpassing til kombinert (2 beste av 3 omganger er tellende): Poengrutinene i class Skihopper må endres/utvides

56