En første innføring i objekter og klasser

3. oktober 2006

Arild Waaler

Et program er en modell av verden

- Et program er en *modell* av et lite utsnitt av verden.
- Et program kan *eksekveres* og dermed *utføre beregninger*: det kan transformere data fra en form til en annen form.

Et programmeringsspråk er laget for å støtte begge disse aspektene.

Med objekt-orientert programmering (OOP) kommer

- en ny måte å representere verden på i et program
- et nytt bilde av hva som skjer under eksekvering
- en metode som effektivt og naturlig leder oss fra en oppgavebeskrivelse til et ferdig program

Inntil nå i kurset:

Programmer som bruker enkle deler av Java:

- Variable av int, boolean, char, double, String
- Arrayer
- Metoder

Modellen av verden = datastruktur + metodenavn!

- Dette er ikke alltid enkelt å uttrykke en modell av verden med bare disse konstruksjonene!
- Spurte du deg *hva du skulle gjøre med datastrukturen* da du løste oblig 2?
- Hva er ditt bilde av hva som skjer når programmet ditt kjører?

Ideen bak OOP

- Konstruer en gruppe av individer der hvert individ tilbyr seg å løse bestemte oppgaver.
- Når de løser sine bestemte oppgaver, kan de spørre andre individer om assistanse.
- Når et individ får forespørselen, får det også ansvaret for å utføre den og full frihet å velge løsning.
- På den måten løses oppgavene kollektivt.
- I OOP-sjargong kaller vi individene for *objekter*.
- Å lage et OOP-program er å opprette en gruppe av objekter og spesifisere hvordan disse kan samarbeide om å løse oppgavene.

OOP gjenspeiler oppgaveløsning fra dagliglivet

- Du går inn på en kina-restaurant, leser menyen, bestiller
- I OOP-begreper: en kelner er et objekt som tilbød deg tjenester ut fra menyen
- Du rettet en forepørsel (bestillingen) sammen med parametre (spesielle ønsker).
- Etter at kelneren hadde akseptert forespørselen påtok kelneren seg ansvaret for å levere varen.
- Hvordan oppgaven løses, er skjult for deg, og det er du heller ikke interessert i å vite!

Et meget stort antall individer har bidratt!

- Kelnerens oppgave er primært å registrere bestillingen og levere varene
- Noe av bestillingen besørger han selv, resten løser han ved å delegere oppgavene videre
- På kjøkkenet blir oppgavene igjen fordelt til mange små oppgaver
- Biffen som inngår i retten kom fra et sted. Den ble trolig bestilt av restauranten og levert av noen
- Biff-bestillingen ble splittet opp i mindre jobber som involverte bønder, slakteriet, slakteren osv.

Oppgaven ble løst kollektivt av en gruppe samarbeidende individer, hver med sine oppgaver!

Bildet kan overføres til OOP

I analogi med dette:

- Vi kan gjøre rede for utføringen av OOP-programmer ved å tenke oss at vi foretar en *simulering*
- Dette er spesielt viktig når dere i inf1010 skal lære om tråder og parallelle Java-programmer
- Når dere bruker klasser som andre har skrevet (for eksempel fra EasyIO) gir dette en godt bilde av programutførelsen

Objekter og klasser

- I OOP-begreper snakker vi om at objektene er av en *klasse*. 'Kinarestaurant' kan være et eksempel på en klasse
- Klassene *definerer* hvordan objektene av klassen oppfører seg, dvs. hvilke *metoder* som klassen inneholder
- Alle objektene av nøyaktig samme klasse skal tilby nøyaktig de samme metodene
- Objektene *utfører handlinger*
- Objektene eksisterer ikke før programmet kjører. De må *opprettes* under kjøring

Objektmetoder og statiske metoder

- Når vi ber et objekt om å utføre en metode, så har metodekallet vårt en *adresse*, nemlig objektet
- En static metode har ikke på samme måte noen adressat
- Mens vi med en static metode tenker oss at *vi gjør noe med* noen data, er det i forhold til objekter ofte lurt å spørre seg: *hva kan de gjøre for meg*?
- Vi tenker da at beskjeden vår har en mottager som påtar seg ansvaret for oppgaven.

Parentes: Interface og subklasser

To andre poenger ved kinarestaurant-eksempelet som også reflekteres i OOP, men som ikke behandles nærmere:

- Alle restauranter tilbyr mer eller mindre de samme tjenestene. Men de tolker samme forespørsel forskjellig. To kinarestauranter kan servere helt forskjellige ting når vi bestiller "vårrull" (polymorfi)
- Vi vet en masse ting om en kinarestaurant simpelthen fordi den er en restaurant
- I Inf1010 lærer dere om Java interface, som kan brukes til å skrive programmere med polymorfi, og om subklasser og arv, som har med det andre punktet å gjøre.

Oppsummering av OOP

Vi oppretter en gruppe av objekter med det formål at

- Hvert objekt skal ha sin bestemte og naturlige oppgave
- Gruppen skal kollektivt samarbeide om å løse oppgavene
- Objektene opprettes som instanser av klasser
- Objektene samarbeider ved å sende meldinger til hverandre i form av å kalle hverandres metoder og overføre informasjon mellom hverandre i form av parametere og returverdier til metoder
- Metodekallene har en entydig mottager som, hvis det aksepterer oppdraget, overtar ansvaret for oppgaven

Fra ideen om OOP til OOP-program

OOP-ideen realiseres i Java i form av klasser og objekter.

- Vi *tenker i form av objekter, men programmer i form av klasser.*
- Programmene deres til nå har bestått av én klasse med en main-metode. Hver kjøring sammenlignes med et objekt, mens programmet selv er en klasse.
- Et OOP-program består normalt av mange klasser, og vi kan tenke oss at hvert objekt av en gitt klasse svarer til en "kjøring av klassen."
- En slik programkjøringene kan samarbeide med alle de andre som det kjenner til, dvs. som det har et navn på, ved metodekall.
- Vi starter ett av programmene ved å kalle klassens main-metode, og deretter må objektene *opprettes* når programmet kjører.

Hva består en klasse av?

Klassenavn
Variable
Konstrukør
Metoder

Datastruktur – der informasjonen lagres

Initialisering av datastrukturen når objektet opprettes

Tjenestene som tilbys

Helhet og deloppgaver i OOP

En stor fordel når vi lager OOP-programmer et at

- *vi trenger ikke å ha oversikt over hele programmet eller hele datastrukturen* når vi skriver en metode.
- Kinarestauranten: 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.
- Slik også når vi programmerer: vi programmerer en klasse ut fra klassens perspektiv og glemmer resten av helheten mens vi gjør dette.

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.

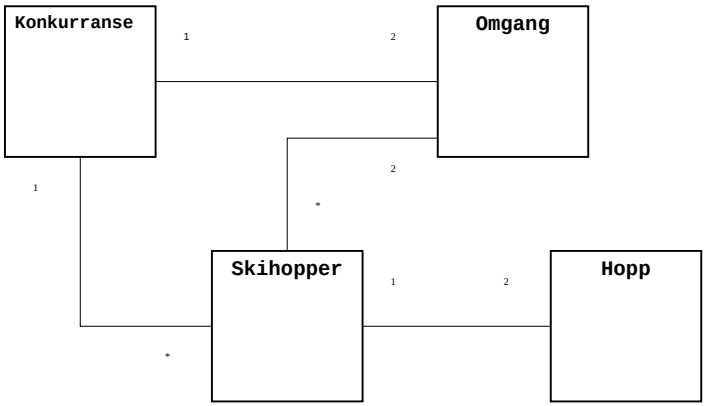
Hva er objektene?

- I modellen av kinarestauranten svarer objektene til konkrete, virkelige individer
- 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*

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

Klassediagram av hopprenn-systemet



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.

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

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
```


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
}
```


void kommandoløkke()

```
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));
```


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

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!

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();
    }
```


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: Morten Dæhlen
  Klubb: UiO

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


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 ();

```
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;
}
```

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!

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

void stokk(...) og Random tall-generator

```
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;
      }
   }
```

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

Her kunne vi unngått å overføre deltager-arrayen som parameter

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 summer

```
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;
  }
```

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.

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.

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