En innføring i objektorientert programmering

Jeg skal nå først forklare ideen bak OOP og illustrere ideen ved et eksempel fra dagliglivet. Så skal jeg vise OOP i bruk. Vi skal, i store skritt, gå fra beskrivelse av en oppgave til utforming av et fullstendig OOP-program i Java. Dere vil da også bli kjent med noen nye Java-ord ved å se dem i bruk.

Et program er en modell av verden

Som navnet antyder, er objektorientert programmering (OOP) en måte å programmere på. OOP er et nytt punkt i Inf1000, og jeg vil først sette det i perspektiv ved å reflektere litt over det vi har lært til nå. Når vi skal forklare hva et program *er*, er det viktig å skille to aspekter fra hverandre:

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

Alle programmeringsspråk er laget for å støtte begge disse aspektene. Et språk må for det første være slik at vi skal kunne uttrykke (eller *representere*) vår modell av verden i det. Dernest skal det kunne *utføres* på en entydig måte, vi må ikke være i tvil om hva som skjer når programmene våre eksekverer.

Inntil nå har vi i kurset sett på et lite utsnitt av Java der vi har variable av grunntyper (int, boolean osv.) og av to mer komplekse typer: String og array. Dessuten har vi lært om løkker, forgreninger og metoder. Dere har akkurat avsluttet en innleveringsoppgave der dere har brukt kun disse delene av Java, og la oss før vi går over på OOP se på slike programmer i lys av de to punktene jeg startet med.

I programmet dere skrev i oblig 2 representerte dere en liten modell av verden, nemlig den som var beskrevet for dere i oppgaveteksten. Denne modellen uttrykte dere for det første i en *datastruktur*, sannsynligvis i form av variable av type String, array, int, char og boolean, og dernest i *valg av metodenavn*. Dere måtte uttrykke modellen deres på denne måten siden dette er alt dere til nå har lært av Java. Min første påstand er at det er ikke rett frem å uttrykke modellen i Java siden verden ikke primært består av arrayer, tall og strenger. Når dere leser en oppgavetekst, vil ordene som brukes for å forklare hva systemet skal brukes til neppe inneholde ordene int, tabell og streng (uten kanskje i hint)! Programmeringsspråket krever at vi transformerer beskrivelsen av hva programmet skal gjøre ganske mye før vi klarer å uttrykke den i datastrukturer og programkode. Valg av metodenavn hjelper oss imidlertid til å lage programmer som, når vi leser dem, er gjenkjennelig ut fra beskrivelsen i oppgaveteksten.

Når du har laget programmet, tenkte du sannsynligvis på *hva du skulle gjøre med datastrukturen* for at programmet skulle løse oppgaven riktig. Du hadde trolig også et bilde av hva som skjer når programmet kjører i bakhodet. Jeg tipper at dette bildet kan sammenlignes med en sekretær som jobber i et kjempestort arkiv, der alt som gjøres er å skrive symboler på ark, ta kopier og legge papirer i mapper. Metodene er bare hendige navn på en bestemt sekvens av operasjoner som sekretæren gjør. Dette er riktignok et ganske godt bilde av hva som skjer inne i datamaskinen når programmet kjører. Problemet er at det ofte ikke gir oss særlig mye hjelp til å forstå hvordan *vi kan programmere en datamaskin* for å løse en oppgave.

OOP kommer oss her til unnsetning. For det første skal vi lære noen nye konstruksjoner i Java som gjør det mulig for oss å uttrykke vår modell av verden i et Java-program på en mye mer naturlig måte enn vi har kunnet til nå. Dernest inviteres vi til å tenke på en bestemt måte når vi strukturerer verden. Den størst utfordringen ved å lære OOP er faktisk å lære å tenke objekt-orientert! Men fordelen er at vi da også lærer en metode for å utvikle programmer som er effektiv og hjelper oss til å skrive programmer med en struktur som svarer til strukturen i beskrivelsen av det i naturlig språk. Det blir dermed liten avstand mellom beskrivelsen og koden, noe som kan gjøre programmet både elegant og intuitivt korrekt. OOP gir oss dessuten et annet bilde av hva som skjer når et program eksekverer, et bilde som er nyttig når vi skal lage kompliserte programmer.

Ideen bak OOP

Dette er 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 aksepterer 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.

Dette er måte å løse oppgaver på som brukes overalt i dagliglivet. Tenk deg at du går inn på en kina-restaurant. Du blir mottatt av en kelner, som viser deg til bordet og kommer med en meny. Du leser menyen, bestiller og etter en stund kommer maten.

I OOP-begreper kan vi se på kelner som et objekt som tilbød deg tjenester ut fra menyen. Du rettet en forespørsel (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: alt du trenger å forholde deg til er de tjenestene som restauranten tilbyr og hvordan du kan gjøre bruk av dem for å løse det som er din oppgave, nemlig å skaffe deg en god porsjon kinamat.

Men hvis vi sporer hva som skjer, ser vi fort at det er en hel rekke andre individer som også bidrar til at bestillingen din utføres. Kelnerens oppgave er primært å registrere bestillingen og levere varene. Noe av bestillingen besørger han selv, for eksempel drikken, mens resten løser han ved å delegere oppgavene videre til kjøkkenet. På kjøkkenet blir oppgavene igjen fordelt til mange små deloppgaver, der noen er ansvarlig for at alt koordineres. Vi kan fortsette enda lenger. Biffen som inngår i retten kom fra et sted. Den ble trolig bestilt av restauranten og levert av en varebil. Underveis ble bestillingen splittet opp i mindre jobber som involverte bønder, slakteriet, slakteren osv. Vi ser at det i virkeligheten er et meget stort antall individer som bidrar til måltidet som serveres på kinarestauranten. *Oppgaven ble løst kollektivt av en gruppe samarbeidende individer, hver med sine bestemte oppgaver*.

En slik kjede av koordinerte handlinger gir også et bilde av hvordan et OOP-program eksekverer. Mens vi til nå i kurset kunne bruke bildet av et stort arkivskap for å illustrere eksekvering av programmer, kan vi gjøre rede for utføringen av OOP-programmer ved å tenke oss at vi foretar en *simulering*. Dette bildet er spesielt viktig når dere i inf1010 skal lære om tråder og parallelle Java-programmer. Det er også nyttig å tenke slik når dere bruker klasser som dere ikke selv har skrevet, slik som dere alt har gjort med klassene In og Out fra EasyIO.

Selv om du ikke har spist på akkurat denne kinarestauranten før, vet du en god del om hva du har i vente. I OOP-begreper snakker vi om at objektene er av en *klasse*. Det er klassene som definerer hvordan objektene av klassen oppfører seg, mens det er objektene som *utfører handlinger*. 'Kinarestaurant' kan være et eksempel på en klasse.

I et OOP-program må vi starte med å finne klassene og spesifisere hvordan typer tjenester objektene av hver klasse skal tilby, dvs. hvilke *metoder* som klassen inneholder. Alle objektene av nøyaktig samme klasse skal tilby nøyaktig de samme metodene. For at metodene skal utføres, må vi imidlertid først *opprette objekter* av klassen, for det er objektene som kan utføre metoder. Klassene skal bare definere hva objektene kan gjøre. Det er mulig å unngå dette skjemaet i Java ved å lage metoder som er `static`, og det er dette dere har gjort til nå i kurset. Inntil nå har vi ikke hatt noe valg siden vi ennå ikke har lært om objekter, mens fra nå av kan vi velge om metodene våre skal være statiske eller tilhøre objekter.

Når vi ber et objekt om å utføre en metode, så har metodekallet vårt en *adresse*, nemlig objektet, og objektet er ansvarlig for utførelsen. En `static` metode har ikke på samme måte noen adressat. Og 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*? Når vi stiller dette spørsmålet, så tenker vi objektorientert. Vi tenker at beskjeden vår har en mottager som, hvis forespørselen aksepteres, påtar seg ansvaret for oppgaven.

Det er to andre poenger ved kinarestaurant-eksempelet som også reflekteres i OOP, men som ikke behandles nærmere i Inf1000. Jeg nevner dem likevel kort:

- 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". I OOP sier vi: et objekt står fritt til å tolke forespørselen. Dette utnyttes i en kraftig programmeringsteknikk som kalles polymorfi.
- Vi vet en masse ting om en kinarestaurant simpelthen fordi den er en restaurant.

I Inf1010 lærer dere om Java grensesnitt (interface), som kan brukes til å skrive programmere med polymorfi, og om subklasser og arv, som har med det andre punktet å gjøre. Gode Java-programmerere utnytter disse teknikkene hele tiden.

Før vi går over til Java-eksempelet, la meg oppsummere. I OOP tenker vi oss at vi oppretter en gruppe av objekter med det formål at hvert objekt skal ha sin bestemte og naturlige oppgave, og at gruppen kollektivt samarbeider om å løse oppgavene som programmet er laget for løse. Objektene opprettes som instanser av klasser. Et objekts oppførsel defineres av dens klasse. Objektene samarbeider ved å rette forespørsler 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 overtar ansvaret for oppgaven.

Fra ideen om OOP til OOP-program

OOP-ideen realiseres i Java i form av klasser og objekter. Når vi lager et program må vi *tenke i form av objekter, men programmere i form av klasser*. Vi kan illustrere dette poenget ved å ta utgangspunkt i programmer av den type dere har laget til nå. Programmene deres har bestått av én klasse med en main-metode. Når programmene kjører, kan 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.

En klasse består av en datastruktur, en konstruktør-metode og alle de andre metodene. Et objekt opprettes når vi skriver `new`, vi skal se eksempler på det under.

Klassenavn	
Variable	Datastruktur – der informasjonen lagres
Konstrukør	Initialisering av datastrukturen når objektet opprettes
Metoder	Tjenestene som tilbys

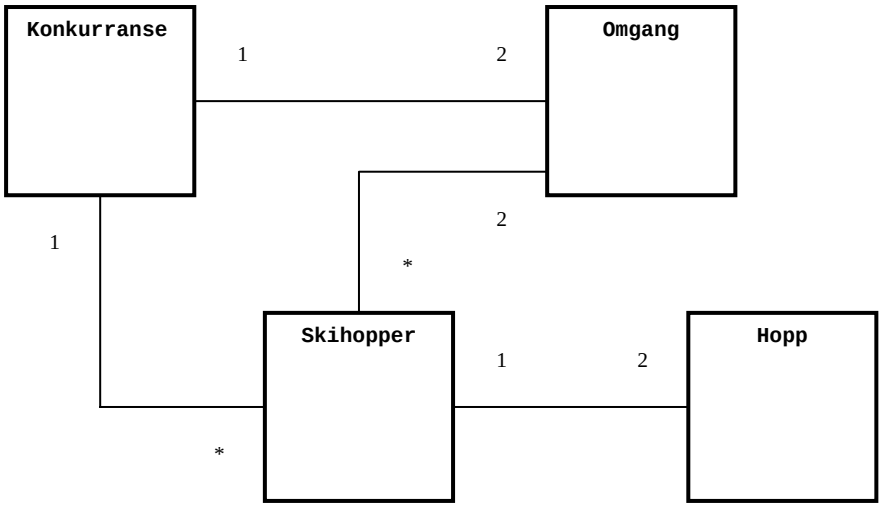
En stor fordel når vi lager OOP-programmer et at *vi trenger ikke å ha oversikt over hele programmet eller hele datastrukturen* når vi skriver en metode. Hvis vi igjen ser på kinarestauranten, kan vi tenke oss at vi ”skifter hatt” og ser systemet gjennom øynene til hver av aktørene for seg. Når vi er 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.

Et system for administrasjon av hopprenn

Vi skal nå lage et objekt-orientert Java-program for å administrere en hopp-konkurranse. Programmet skal registrere navn og idrettslag til skihoppere. Det skal trekke startlisten til første omgang, lese inn lengde og 5 stilkarakterer for hvert hopp, beregne poengsum og skrive ut resultatlisten. I 2. omgang skal startrekkefølgen være omvendt av resultatlisten fra 1. omgang. Metoden for beregning av poengsum ut fra lengde og stilkarakterer er beskrevet i oppgave 5.9 i læreboka.

Modellen av kinarestauranten er spesielt intuitiv fordi objektene i modellen svarer 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*, eventuelt kan vi se for oss at det er en rennansvarlig som står for registreringen av alle data.

Legg merke til at typen til objektene svarer til sentrale substantiver i teksten! En god tommelfingerregel er å merke seg substantivene, for disse gir ofte opphav til en god klasse-inndeling. Dette er ikke tilfeldig: en av styrkene til OOP er at man kan lage programmer med en klasseinndeling som samsvarer med begrepene som brukes i en beskrivelse av problemet i naturlig språk. Vi bør alltid angripe en slik programmeringsoppgave med å identifisere klassene og tegne dem i et enkelt klassediagram, og velger vi de 4 uthevede ordene som våre klasser kan vi ha følgende diagram:



Strekene angir et direkte forhold mellom objekter av klassene. Tallene angir antall som står i slike forhold, og skal leses slik. Ta først linjen mellom Konkurranse og Omgang: Én konkurranse består av 2 omganger (2-tallet til høyre over linjen), mens hver omgang tilhører én konkurranse (1-tallet til venstre over linjen). Hver konkurranse består av 0 eller flere skihoppere (angitt med *), mens hver skihopper tar del i én konkurranse og i 2 omganger, osv.

Konkurranse-klassen er den "øverste" klassen, den klassen som styrer det hele og den første vi støter på i main-metoden. Hvis vi antar at klassen som main-metoden ligger i heter Hopprenn, trenger vi ikke mer kode i denne klassen enn dette:

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

Anta at vi kompilerer (`>javac Hopprenn`) og kjører programmet (`>java Hopprenn`). Når vi utfører main-metoden, skjer følgende:

1. Det opprettes et objekt av klassen `Konkurranse` som heter `rennadm`. Dette skjer idet setningen `new Konkurranse()` utføres.
2. Når objektet `rennadm` opprettes, utføres en bestemt metode i `class Konkurranse` som kalles *konstruktøren*. Konstruktøren heter alltid det samme som klassen – den skrives uten typeangivelse – og utføres en og bare en gang for hvert objekt (nemlig når det opprettes). Konstruktøren i Konkurranse-klassen heter derfor `Konkurranse().`
3. 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.`

Vi følger utviklingen av programmet og tar nå for oss konkurranse-klassen. Likesom kelneren gir kundene en meny, kan vi angi hvilke tjenester en slik klasse skal tilby ved å bestemme menyen på skjermbildet som brukeren kan velge fra:

```
*** 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
```

Punkt 6 er lagt til for å gjøre programmet litt morsommere, og er dessuten hendig for feilsøking. Skisse av koden i Konkurranse-klassen med konstruktøren og kommandoløkke-metoden:

```
import easyIO.*;
class Konkurranse {

    Konkurranse() {
        System.out.println("HOPP-PROGRAM VERSJON 1.0");
    }

    void kommandoløkke(){
        skrivMeny();
        int valg;
        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));
    }
    // Her kommer alle de andre metodene i class Konkurranse
}
```

Konstruktøren gjør ikke stort, den skriver ut en liten melding til skjerm. Oppgaven til konstruktøren er normalt å gi datastrukturen riktig startverdi (dvs. å *initialisere* den). Konstruktøren kan fjernes helt fra koden vår om vi ikke trenger den til noe. I kommandoløkken har jeg skrevet navnet på noen metoder som skal sørge for at kommandoen utføres. Merk at mens vi skrev `rennadm` foran kallet på kommandoløkken, står det ingenting foran for eksempel kallet `registrerDeltager()`. Når det ikke står noen adressat for metodekallet, betyr det at adressaten er det aktive objektet, i dette tilfellet `rennadm`.

Merk også at vi nå jobber oss ”utenfra inn” i programmet! Vi starter med å identifisere de metodene som først vil bli kalt og navngir disse før vi egentlig har tenkt over hvordan de skal

kodes. Når vi nå skal kode `registrerDeltager()` blir vi nødt til å tenke på hvilke objekter og metoder denne trenger, og da navngir vi disse. Vi oppretter med andre ord objekter og identifiserer metoder etter hvert som vi trenger dem og før vi har skrevet kode for dem. Dette er ofte en god metode å følge også når vi skal *skrive* programmer.

Vi må nå ha mulighet til å lagre informasjon om hver hopper og må bestemme oss for datastruktur. For hver hopper oppretter vi et nytt objekt av klassen `Skihopper` og legger disse inn i en array på enklest mulig måte. Her er en datastruktur:

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

Når vi skal lage et nytt Skihopper-objekt, må vi lese inn navn og idrettslag fra tastaturet. I god OOP-stil kan vi delegere denne oppgaven til skihopper-objektet, dvs. skihopperen får ansvar for å hente informasjon om seg selv! Vi legger denne jobber til konstruktøren for Skihopper-objektet. Dette gir oss enkel kode for `registrerDeltager()` i Konkurranse-klassen:

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

Skihopper-klassen inneholder bl.a. følgende kode:

```
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();
    }


    public String toString(){
        String s = "";
        if( startnr != UDEFINERT ) s += startnr + " ";
        s += navn +" "+ idrettslag;
        return s;
    }
    // her kommer alle de andre metodene i class Skihopper
}
```

Metoden `toString()` er hendig fordi det er den som kalles når vi skriver ut et *objekt* av klassen `Skihopper`. I klassen `Konkurranse` har vi følgende metode som illustrerer dette:

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

Husk at `deltager[i]` er et Skihopper-objekt. Når vi skriver ut objektet, kaller vi altså dennes `toString`-metode, med den effekt av vi skriver ut startnummeret hvis vi har foretatt trekning, ellers skriver vi bare ut navnet og klubben. Selve trekningen utføres av en metode i Konkurranse-klassen. Det første vi gjør er å minke størrelsen på deltager-arrayen, dette gjøres for å få enkel kode (en bedre løsning er å bruke ArrayList!).

```
    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;
    }
```

Jobben gjøres i metoden `stokk`. Merk også at vi har en boolsk variabel `trukket` som settes til true. Dette er fordi jeg vil hindre at man går til første omgang før man har trukket startnumre, se koden under. Når vi skal trekke, gjør vi bruk av en nyttig metode fra Java-biblioteket som genererer tilfeldige tall. Metoden ligger i klassen `Random` i pakken `java.util` og heter `nextInt`. Hvis vi har et Random-objekt `tall` og skriver `tall.nextInt(23),` så returner metoden et tilfeldig tall større eller lik 0 og mindre enn 23. Vi skal bruke den til å stokke om på elementene i deltager-arrayen, og da må vi først opprette et objekt av klasse `Random` og så kalle dennes `nextInt`–metode.

```
import java.util.*;

    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;
        }
    }
```

Nå er vi klare til å kikke på klassen `Omgang`. I `class Konkurranse` deklarerer vi to variable av denne klassen:

```
Omgang førsteOmgang, andreOmgang;
```

Initielt har disse en spesiell Java-verdi som heter `null`. Det betyr at det ikke finnes noen objekter som disse to variablene refererer til. Omgangene opprettes og kalles fra Konkurranse-klassens kommandometode:

```
  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;
```

Et omgangsobjekt må vite om hvorvidt det er en første eller annen omgang, så dette overføres til konstruktøren for Omgang-klassen via en boolsk verdi: true for første omgang, false for andre. Vi overfører også startrekkefølgen for omgangen som en parameter til konstruktøren. Metoden `reverser` returnerer en array med elementer i omvendt rekkefølge og er enkel å kode.

Vi presenterer `class Omgang` på samme måten som Konkurranse-klassen:

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

Klassen har en kommandoløkke-metode med samme oppbygging som for Konkurranse-klassen. I denne klassen gjør konstruktøren den som den vanligvis gjør, nemlig å initialisere datastrukturen:

```
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];
    }
…}
```

Vi har to arrayer: en for startlisten og en for rekkefølgen så langt. Denne bygger vi opp etter hvert som vi registrerer hopp slik at den alltid er sortert på poengsum. Merk spesielt ordet `this`, for eksempel i `this.startliste = startliste.` Ordet betyr av vi angir det aktive objektet: `this.startliste` betyr derfor variabelen deklarert i klassen, mens verdien `startliste` som den tilordnes er verdien som er argument til konstruktøren.

Den meste interessante operasjonen som tilbys av Omgang-objektene er registrering av nye hopp. Siden programmet også tilbyr automatisk generering av hopp, har jeg delt registreringen inn i to metoder. Den første metoden delegerer oppgaven videre til skihoppere-objektet: skihopperen får selv ansvar for å skaffe data om sitt eget hopp. I den andre metoden sørger vi for at arrayen `rekkeflg` holdes sortert på poengsum. For å få tak i poengsummen, spør vi også skihopperen om assistanse gjennom metoden `poengsum`. Argumentet `førsteomgang` er nødvendig siden skihopperen vil levere ulik poengsum etter henholdsvis første og andre omgang. Det er imidlertid Skihopper-objektets ansvar å tolke forespørselen og å utføre den. I koden under, husk at `førsteomgang` er en variabel som enten er true eller false.

```
    void nesteHopp(){
        if( antall >= startliste.length ) return;
        startliste[antall].nyttHopp(førsteomgang);
        oppdaterListe();
    }

    void oppdaterListe(){
        int i = antall;
        Skihopper aktiv = startliste[antall];
        while( i>0 && rekkeflg[i-1].poengsum(førsteomgang) <
                              aktiv.poengsum(førsteomgang) ){
            rekkeflg[i] = rekkeflg[i-1];
            i--;
        }
        rekkeflg[i] = startliste[antall++];
        System.out.println( "  "+ aktiv.poengsum(førsteomgang) +"
                           poeng  [nr. "+ (i+1) +"]\n" );
    }
```

Det gjenstår nå å kikke på `class Hopp`. Koden for denne følger skjemaet fra de andre klassene og skulle nå være rett frem. Merk at vi her har to konstruktører. Den ene er uten parametere og brukes når vi skal lese inn hopp-data manuelt. Den andre tar inn alle disse data som argumenter og brukes når vi genererer disse data automatisk.

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

    Hopp() {
        // Leser inn lengde og stil og beregner poengsummen
    }

    Hopp( double lengde, double[] karakter ){
        this.lengde = lengde;
        this.karakter = karakter;
        poeng = Poengberegning.poengsum(lengde, karakter);
    }
}
```

Koden er vedlagt – det kan være instruktivt å lese nøye gjennom det hele for å få oversikten. Når du har oversikt over hvordan et OOP-program er laget, er det nemlig mye lettere å skrive

et selv etterpå. Det meste i programmet er forklart i teksten, og det du måtte lure på ut over dette blir forklart senere i kurset!

Metoden som beregner poengsum har jeg lagt i en klasse som heter `Poengberegning`. I denne klassen står det `static` foran alle variable og metoder, og det betyr at metodene og variablene tilhører *klassen* og ikke objekter av klassen. Vi trenger da ikke å opprette noe objekt av klassen før vi kaller metoden, jvf. setningen

```
poeng = Poengberegning.poengsum(lengde, karakter);
```

Her står klassenavnet, og ikke navnet på et objekt, foran metodenavnet. Dette betyr at metodekallet heller ikke har en adressat. Slike metoder kan være nyttige for hjelpefunksjoner som ikke inngår sentralt i den objekt-orienterte modellen.

Oppsummering

I starten så vi på en hverdagssituasjon fra en kinarestaurant som et abstrakt eksempel for å illustrere prinsippet bak OOP, en situasjon som OOP-begrepene viste seg velegnet til å beskrive. Etter dette gikk vi rett over til å løse en konkret Java-oppgave. Men lar hopprenn-programmet seg beskrive på samme naturlige måte som kinarestauranten?

I hopprennet utspenner vi også en liten verden, en gruppe av objekter med dedikerte oppgaver. Konkurranse-objektet `rennadm` har til oppgave å administrere hele gruppen av objekter og kommunisere med brukeren. Objektet er ansvarlig for opprettelse av alle Skihopper-objektene og ordne dem i arrayer, samt for opprettelse av de to Omgang-objektene. Omgang-objektene delegeres ansvar for å administrere det som skjer innen hver omgang. De delegerer selv ansvaret for opprettelse av Hopp-objekter videre til de respektive Skihopper-objektene. Oppgaven løses ved at en gruppe objekter opprettes og samarbeider om løsningen. Samarbeidet reflekteres i metodekall der hver metode har en bestemt adressat. Informasjon overføres i form av parametere og returverdier til metoder.

Vi fant klasseinndelingen til hopp-programmet fra substantivene i oppgaveteksten. Dette er en metode som ofte fungerer godt og reflekterer at klassene i et OOP-program bør svare til begreper det er naturlig å bruke når situasjonen vi skal modellere skal beskrives i vanlig språk.

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! Merk at dette gir dere en metode dere kan følge som tar dere fra oppgavetekst til program i små, naturlige steg.

Underveis må vi velge datastruktur. I eksempelet lagrer hvert hopp informasjon om lengde og stil, skihopperen lagrer informasjon om navn og idrettslag og pekere til to hopp og hver omgang lagrer sekvenser av skihoppere i arrayer, samt informasjon om seg selv (om det er førsteomgang eller ikke). Konkurranseobjektet samler alle hopperne i en array og har pekere til hver omgang.

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? I det øyeblikk du fokuserer på hva du skal gjøre med datastrukturen, står du i fare for å miste objekt-orienteringen av syne, og programmet blir ikke så naturlig og godt som det kunne ha blitt. 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.

Jeg slutter med kort å peke på et fortrinn ved OOP-programmer: de består ofte av moduler med klart avgrenset funksjon. Dette er bra hvis vi ønsker å endre eller forbedre programmet, siden det da er avgrensede deler av det som må endres. Dernest gjør dette at programmene ofte kan utvides uten hele konstruksjonen må lages på nytt. Øvingsoppgavene under illustrerer dette poenget.

Programmeringsoppgaver

A. Enkle utvidelser for mer robust program

1. Legg inn test slik at man ikke kan registrere ny deltager etter trekning.
2. Legg inn test slik at man ikke kan gå til 2. omgang før 1. omgang er avsluttet.
3. Legg inn sjekk på at stilkarakterene er på lovlig format, dvs. mellom 0 og 20 med mulige halvkarakterer.
4. Skift representasjon av poengsum slik at du unngår avrundingsfeilene pga. double-typen.
5. Skriv data underveis til fil slik at data ikke går tapt ved strømbrudd.
6. I Poengberegning er det vist eksempler på modifikatorene public og private. Gå gjennom hele koden og sett inn public/private på alle metoder og variable slik at mest mulig blir private. Du må da skrive get/set-metoder for å aksessere datastrukturen til en klasse fra en annen klasse, se seksjon 8.13 i læreboka.

B. Utvidelser som ikke krever større endringer av programkonstruksjonen

1. I kombinert hopper man 3 omganger og har regler for hvilke to hopp som skal telle. Vi må opprette et nytt Omgang-objekt og tilpasse metoden som regner ut poeng slik at 3. omgang følger kombinert-reglene.
2. I et kretsrenn for gutter og jenter arrangerer man oftest mange ulike renn: ett for hver årsklasse og hvert kjønn. Opprett bare et Konkurranse-objekt for hvert renn.

C. Utvidelser som krever noe større inngrep

1. Hvis vi ønsker å lage statistikk for hver dommer, er det også mulig med den strukturen vi har nå, men det beste er nok å lage en ny `class Dommer`. Finn ut hvor ofte hver dommer dømte en karakter som var lik den som ble strøket fordi den var for lav, henholdsvis for høy. Finn hver dommers gjennomsnittskarakter og den totale gjennomsnittskarakteren.
2. Vi kan nå ikke endre noen data. Modifiser programmet slik at vi kan endre inntastede data, både for skihoppere og hopp.
3. Vi kan risikere at en hopper ikke møter til start. Foreslå en løsning slik at programmet kan håndtere denne situasjonen.

Fullstendig Java-kode for hopprenn-programmet

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

class Konkurranse {
    final int MAX_ANTALL = 60;
    Skihopper[] deltager = new Skihopper[MAX_ANTALL];
    int antallHoppere = 0; // Brukes som indeks i deltager
    Omgang førsteOmgang, andreOmgang;
    Boolean trukket = false; // true når alle har fått startnummer
    Random tall = new Random();
    In tast = new In();

    Konkurranse() {
        System.out.println("HOPP-PROGRAM VERSJON 1.0");
    }

    void skrivMeny(){
        System.out.println();
        System.out.println("*** MENY ***");
        System.out.println("0. Avslutt");
        System.out.println("1. Registrer ny deltager");
        System.out.println("2. Trekning av startnummer");
        System.out.println("3. List alle deltagere");
        System.out.println("4. Første omgang");
        System.out.println("5. Andre omgang");
        System.out.println("6. Generer fiktive deltagere");
        System.out.println();
    }

    void kommandoløkke(){
        skrivMeny();
        int valg;
        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: 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;
                case 6: autogenerer(); break;
                case 9: skrivMeny(); break;
                default: System.out.println("Du tastet feil");
            }
        } while (!(valg == 0));
    }

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

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

```
    /**
     * Stokker rekkefølgen i arrayen av deltagere og registrerer startnummeret
     * i Skihopper-objektene.
     */
    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;
    }

    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;
        }
    }

    Skihopper[] reverser( Skihopper[] tabell ){
        int lengde = tabell.length;
        Skihopper[] ny = new Skihopper[tabell.length];
        for( int i=0; i<tabell.length; i++ )
            ny[--lengde] = tabell[i];
        return ny;
    }

    void autogenerer(){
        System.out.print("Antall automatisk genererte deltagere? ");
        int valg = tast.inInt();
        for( int i=0; i<valg ; i++ ){
            Skihopper ny = genererNyHopper();
            deltager[antallHoppere++] = ny;
        }
        System.out.println( valg + " deltagere lagt til\n");
    }

    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 );
    }

} // slutt class Konkurranse
```

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

class Skihopper {
    private static final int UDEFINERT = -1;
    String navn,idrettslag;
    int startnr = UDEFINERT;
    Hopp førstehopp, andrehopp;
    Random tall = new Random();

    Skihopper( String navn, String idrettslag ){
        this.navn = navn;
        this.idrettslag = idrettslag;
    }

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

    double poengsum(boolean førsteomgang){
        if( førsteomgang )
            return førstehopp.poeng;
        else return førstehopp.poeng + andrehopp.poeng;
    }

    void skrivResultat(boolean førsteomgang){
        System.out.print( navn +" "+ idrettslag +"  "+ poengsum(førsteomgang));
        if( førsteomgang ) System.out.println();
        else System.out.println("  ("+ poengsum(!førsteomgang) +")");
    }

    void nyttHopp(boolean førsteomgang){
        System.out.println("  Nå hopper "+ navn);
        Hopp h = new Hopp();
        if( førsteomgang ) førstehopp = h;
        else andrehopp = h;
    }

    double[] testkarakter = {14.5, 15, 16, 17, 17.5, 18, 18.5, 19, 19.5};
    double[] testlengde = {114, 115, 116, 117.5, 118, 118.5, 119, 120, 121, 122, 122.5, 127};

    void simulerHopp(boolean førsteomgang){
        pause(500);
        System.out.println(" "+ this +": ");
        pause(800);
        double lengde = testlengde[tall.nextInt(testlengde.length)];
        System.out.print(" "+ lengde +"m  ");
        pause(1000);
        double[] stil = new double[5];
        for( int i=0; i<5; i++ ){
            stil[i] = testkarakter[tall.nextInt(testkarakter.length)];
            System.out.print(" "+ stil[i]);
        }
        pause(1000);
        Hopp h = new Hopp(lengde, stil);
        if( førsteomgang ) førstehopp = h;
        else andrehopp = h;
    }

    void pause(int ms){
        try{ Thread.sleep(ms); } catch(Exception e){}
    }

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

```
import easyIO.*;

class Omgang {
    Skihopper[] startliste;
    Skihopper[] rekkeflg;
    int antall = 0;;   // 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];
    }

    void skrivMeny(){
        System.out.println();
        System.out.print("*** MENY ");
        if( førsteomgang ) System.out.println("1. OMGANG ***");
        else  System.out.println("2. OMGANG ***");
        System.out.println("0. Tilbake til hovedmenyen");
        System.out.println("1. Registrer nytt hopp");
        System.out.println("2. List gjenstående hoppere");
        System.out.println("3. Resultatliste");
        System.out.println("4. Simuler resten av omgangen");
        System.out.println();
    }

    void kommandoløkke(){
        In tast = new In();
        int valg;
        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));
    }

    void nesteHopp(){
        if( antall >= startliste.length ) return;
        startliste[antall].nyttHopp(førsteomgang);
        oppdaterListe();
    }

    void oppdaterListe(){
        int i = antall;
        Skihopper aktiv = startliste[antall];
        while( i>0 && rekkeflg[i-1].poengsum(førsteomgang) < aktiv.poengsum(førsteomgang) ){
            rekkeflg[i] = rekkeflg[i-1];
            i--;
        }
        rekkeflg[i] = startliste[antall++];
        System.out.println( "  "+ aktiv.poengsum(førsteomgang) +" poeng  [nr. "+ (i+1)
                            +"]\n");
    }

    void skrivResultat(){
        int nr = 0;
        for( int i=0; i<antall; i++){
            if( i==0 || rekkeflg[i-1].poengsum(førsteomgang) !=
                        rekkeflg[i].poengsum(førsteomgang))
                nr = (i+1);
            System.out.print("Nr "+ nr +". ");
            rekkeflg[i].skrivResultat(førsteomgang);
        }
    }
```

```
    void skrivGjenstående(){
        for( int i=antall; i<startliste.length; i++)
            System.out.println("[Hopper nr. "+ (i+1) +"] "+ startliste[i]);
    }

    void simulerOmgang() {
        while( antall<startliste.length ){
            startliste[antall].simulerHopp(førsteomgang);
            oppdaterListe();
        }
    }

} // slutt class Omgang


import easyIO.*;

class Hopp {
    double lengde;
    double[] karakter;
    double poeng;

    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);
    }

    Hopp( double lengde, double[] karakter ){
        this.lengde = lengde;
        this.karakter = karakter;
        poeng = Poengberegning.poengsum(lengde, karakter);
    }
}

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;
    }

    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;
    }
}
```