Kapittel 5:
Objektkommunikasjon

Redigert av:

Khalid Azim Mughal (`khalid@ii.uib.no`)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.
ISBN: 82-02-24554-0
`http://www.ii.uib.no/~khalid/jfps3u/`
(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

- Ansvar og roller
- Lage gode abstraksjoner
- Strukturert programmering
- Gjentakelse i programkode
- Kommunikasjon og samarbeid
- Forbindelser
- Objekteierskap
- Innkapsling
- Klasseutforming
- Metodeoverlasting
- Dokumentering av kildekode
- Eksempel: CD-samling
- Generert dokumentasjon (`javadoc`)

Ansvar og roller

- Egenskapene og operasjonene definert for en klasse bestemmer:
  - Hvilken abstraksjon klassen representerer.
  - Ansvarsområdet til klassen.
  - Hvilke roller objekter av klassen kan oppfylle.
  - Hvilken programkode som bør inngå i klassen.
- Fordeler med gode abstraksjoner:
  - Enklere å forstå programmet.
  - Enklere å gjøre endringer.
  - Kan redusere størrelsen på programkoden.
  - Blir lettere å identifisere og huske hva hver del av programkoden gjør.
  - Programkoden blir mer generell og gir mulighet for å bruke samme programkode for flere programmer, m.a.o. bidrar til gjenbruk av programkode.
- Nytten av gode abstraksjoner blir tydeligere etter hvert som størrelsen på programmene øker.

Lage gode abstraksjoner

- Hva er en god abstraksjon? Gode abstraksjoner:
  - Binder sammen egenskaper og operasjoner som hører sammen.
  - Gjør det mulig å unngå gjentakelse i programkoden.
  - Begrenser omfanget av fremtidige endringer.
  - Gjør programmet enklere å forstå:
    - Gir klasser og objekter med lett forståelige ansvarsområder og roller.
    - Skjuler detaljer som ikke er nødvendig for den overordnede forståelsen.
- Vurder hvilke metoder som vil være nyttige å ha for å arbeide på feltene i en klasse.
- Fjern eller skjul metoder og felt som ikke er til direkte nytte i en klasse.
- Ikke bland sammen mange ansvarsoppgaver. La rollen til hvert objekt være så klar som mulig.

- Abstraksjon til et objekt kommer ofte før avgjørelser om dets *implementasjon*.
- Men implementeringsprossessen kan gi deg innsikt og ideer om nye abstraksjoner som kan være nyttige.

Programmeringsmetodologi: *strukturert programmering*

- *Innmaten* til objekter utformes ved å anvende *strukturert programmering*.
- Handlinger i metoder beskrives kun v.h.a. følgende *kontrollstrukturer*:
  - Sekvens (blokker)
  - Valg (betinget utføring)
  - Repetisjon (løkker)
- Fører til metoder som er enklere
  - å forstå,
  - å teste og avluse,
  - å modifisere og vedlikeholde.

Gjentakelse i programkode

- Gjør vedlikehold vanskeligere
- Oppdateringer må gjøres parallelt på flere steder.
- Tyder på at programkoden bør omstruktureres.

Eksempel:

```
> java AntallRapport1
Antall biler: 5
Antall båter: 1
Antall tog: 3
Antall mopeder: 2
5 biler, 1 båt, 3 tog og 2 mopeder.

import java.util.Scanner;
public class AntallRapport1 {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Antall biler: ");
    int antall_biler = tastatur.nextInt();
    System.out.print("Antall båter: ");
    int antall_båter = tastatur.nextInt();
```

```
    System.out.print("Antall tog: ");
    int antall_tog = tastatur.nextInt();
    System.out.print("Antall mopeder: ");
    int antall_mopeder = tastatur.nextInt();

    System.out.print(antall_biler);
    if (antall_biler == 1) {
      System.out.print(" bil, ");
    } else {
      System.out.print(" biler, ");
    }

    System.out.print(antall_båter);
    if (antall_båter == 1) {
      System.out.print(" båt, ");
    } else {
      System.out.print(" båter, ");
    }

    System.out.print(antall_tog);
    System.out.print(" tog og ");

    System.out.print(antall_mopeder);
```



```
    if (antall_mopeder == 1) {
      System.out.println(" moped.");
    } else {
      System.out.println(" mopeder.");
    }
  }
}
```

- Totalt: 38 linjer kode.
- Hver ny transporttype vil kreve ca. 8 nye linjer og berører 3 gamle linjer.
- Hvis spørreform skal endres, til f.eks. "Hvor mange biler?", så må endringen gjøres over flere linjer.
- Enkelt å gjøre feil. F.eks hvis man kopierer koden til *mopeder* for å behandle *busser*, og glemmer å oppdatere sannhetsuttrykket.

```
System.out.print(antall_busser);
if (antall_mopeder == 1) {
    System.out.println(" buss.");
} else {
    System.out.println(" busser.");
}
```

- Hvordan unngå gjentakelse:
  - Bruk løkker.
  - Trekke ut felles atferd i egne metoder.
  - Binde oppførsel og egenskaper sammen.
  - Bruke tabeller i stedet for å behandle elementer separat.

Kode med mindre gjentakelser og bedre abstraksjoner

- Introduserer en ny abstraksjon for å forenkle koden: `Antall`.

```
import java.util.Scanner;
class Antall {
  String entall, flertall;
  int antall;
  Antall(String entall, String flertall) {
    this.entall   = entall;
    this.flertall = flertall;
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Antall " + flertall + ": ");
    antall = tastatur.nextInt();
  }
  void skrivUt() {
    if (antall == 1) {
      System.out.print("1 " + entall);
    } else {
      System.out.print(antall + " " + flertall);
    }
  }
}
```

```
// Klient som bruker Antall-klassen.
public class AntallRapport2 {
  public static void main(String[] args) {
    Antall[] alle = { new Antall("bil", "biler"),
        new Antall("båt", "båter"), new Antall("tog", "tog"),
        new Antall("moped", "mopeder") };
    for (int i=0; i < alle.length; ++i) {
      alle[i].skrivUt();
      int resterende = alle.length - 1 - i;
      switch (resterende) {
        case 1:  System.out.print(" og "); break;
        case 0:  System.out.println(".");  break;
        default: System.out.print(", ");
}}}}
```

- Totalt: 30 linjer kode.
- Nøyaktig lik atferd.
- Å legge til en ny transporttype vil koste under én ny linje og berører ingen gamle linjer.
- Å endre spørreform krever kun endring i en linje:

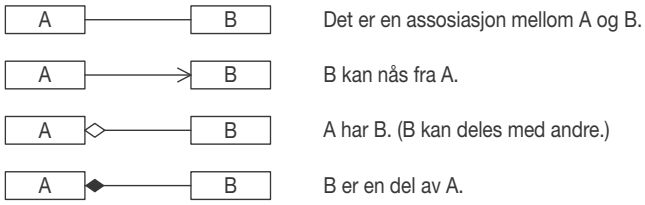
```
System.out.print("Hvor mange " + flertall + "? ");
```


Kommunikasjon og samarbeid

- Metodekall brukes i Java for å
  - Få objekter til å samarbeide.
  - Spørre objekter om informasjon.
  - Gi informasjon til andre objekter.
  - Delegere arbeid til andre objekter.
  - Flytte informasjon mellom objekter i form av parametere og returverdier.

- For å kalle en metode på et objekt trenger man en referanse til objektet.
- Hvordan få tak i en objektreferanse?
  - Opprette et nytt objekt med `new`-operatoren. Hjelper ikke hvis man ønsker å ta kontakt med et objekt som allerede har blitt opprettet.
  - Be et annet objekt om objektreferansen. Men hvordan får man tak i objektreferansen til objektet man skal spørre?
  - Ta vare på referansen i en lokal variabel. Referansen går tapt når metoden avslutter.
  - Ta vare på referansen i en feltvariabel. Dette oppretter en langvarig forbindelse til det andre objektet.

Forbindelser: UML-notasjon



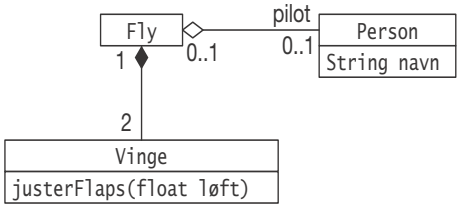
Forholdstall	Beskrivelse
1	Ett objekt inngår i hver forbindelse.
0..1	Ett eller ingen objekter inngår i hver forbindelse.
0..*	Et hvilket som helst antall objekter inngår i hver forbindelse.
n	*n* antall objekter inngår i hver forbindelse.
0..*n*	Opp til og med *n* antall objekter inngår i hver forbindelse.

Eksempel på forbindelser

```
class Person {
    String navn;
    // ...
}

class Vinge {
    void justerFlaps(float løft) {
        //...
    }
    // ...
}

class Fly {
    Vinge venstreVinge;
    Vinge høyreVinge;
    Person pilot;
    // ...
}
```

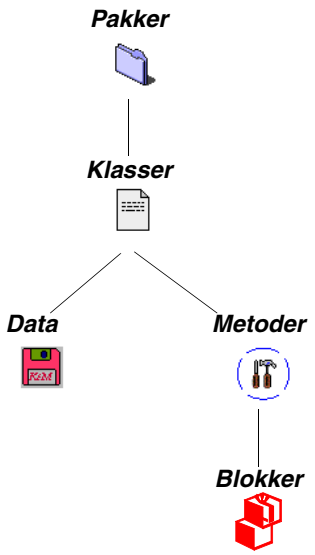


- Ett fly har (eier) to vinger.
- En vinge er tilknyttet ett fly.
- Ett fly kan ha én person (om gangen) som pilot.
- En person kan være en pilot for ett fly (om gangen).

Objekteierskap

- Eieren av et objekt har ofte/vanligvis
  - en feltvariabel med en referanseverdi som peker til det eide objektet.
  - hovedansvaret for å vedlikeholde en referanse til objektet.
  - kontroll over levetiden til objektet.

Innkapsling



- *Abstraksjon* fokuserer på synlig atferd til et objekt, mens *innkapsling* fokuserer på å skjule implementasjon som gir denne atferden.

  En klasse har 2 utsnitt (*views*):
  - *Kontrakt* som tilsvarer vår abstraksjon av atferd som er felles for alle objekter av klassen.
    - Består av navn på metoder og dokumentasjon om hvordan *klienter* (dvs. andre objekter) kan bruke objekter av klassen, og hvordan hver metode burde brukes.
    - Formål: tillate objekter å kommunisere med hverandre.
  - *Implementasjon* tilsvarer *representasjon* av abstraksjonen og *mekanismer* som gir den ønskede atferden.
    - Innholder feltvariabler og Java-setninger som gir oppførsel når de utføres.

Skjuling av informasjon

- *Innkapsling* fører til separasjon av kontrakt og implementasjon.
  - Objektet betraktes som en *svart boks* der *innmaten* er skjult.
    Kontrakt:
    - **HVA** klassen tilbyr - klienter utnytter atferden til klassen.

    Implementasjon:
    - **HVORDAN** klassen implementerer sin atferd - angår ikke klienter.

- Skjuling i Java:
  - Hele klasser kan skjules
  - Individuelle felt og metoder i en klasse kan skjules.
  - Adgangsmodifikatorer brukes i deklarasjonen for å bestemme skjuling.

Modifikator	Effekt
`public`	Klasser, felt og metoder er tilgjengelig for alle.
`private`	Felt og metoder er kun tilgjengelige for programkode innenfor klassen de tilhører. Klasser kan ikke være private.

Data-abstraksjon vha. *skjuling av informasjon*

- Skjul implementasjonsdetaljene til en klasse:
  - Bruk adgangsmodifikatorer for å skjule felt og metoder.
  - Enklere og mer oversiktlig kontrakt.
  - Tillater endring av implementasjon uten behov for endringer i klienter (dersom kontrakten til klassen ikke er endret).
  - Fører til mindre avhengighet mellom programdeler.
  - Oppfordrer til gjenbruk.
  - Resulterer i programdeler som er *svarte bokser*.

Klasseutforming

- Bruk *standard* format for klassedeklarasjoner.
- Del opp klasser med for mange egenskaper eller for mye ansvar/atferd.
- Bruk meningsfylte navn for klasser og metoder som gjenspeiler deres ansvar.
- Feltvariabler:
  - Tilhører som oftest implementasjonen, fremfor kontrakten, og bør defineres som private.
  - Bør initialiseres i konstruktører.
  - Kan være en ren implementasjonsdetalj og trenger ikke alltid mutatorer og/eller selektorer.
- Initialiser alltid lokale variabler i metoder.
- Ikke bruk for mange primitive datatyper i en klasse.

Metodeoverlasting

- *Signatur* til en metode består av *metodenavn* og *formell-parameter-listen* til metoden.
- Hvilken metode som blir utført avhenger bl.a. av metodens signatur.
- Ved kall av en metode til et objekt, må det finnes kun én metode som har en signatur som matcher kallet.
- Konstruktøroverlasting er en form for metode-overlasting.
- Endring av returverdien *alene* kan *ikke* brukes til å oppnå metodeoverlasting.

```
class Lys {
    public double kostnad(int antallTimer) {
        double kWh_pris = 0.35; // øre pr. kiloWatt-time
        double pris = ((double)antallWatt * antallTimer/ 1000)* kWh_pris;
        return pris;
    }
    public double kostnad(int antallTimer, double kWh_pris) {
        double pris = ((double)antallWatt * antallTimer/ 1000)* kWh_pris;
        return pris;
    }
    ...
}
```

signatur

Dokumentering av kildekode

- Overordnet mål: Et program bør være oversiktlig og lett å lese.
- Hva er viktig å dokumentere:
  - Det som ikke er opplagt.
  - Metoder: hva de gjør, hvordan er mindre viktig.
  - Felt: hva de brukes til
  - Klasser:
    - Hvilken abstraksjon klassen representerer og hvilket ansvarsområde den har.
    - Hvilken rolle objekter oppfyller og hvordan objektet typisk blir brukt.
  - Algoritmer som blir brukt. (pseudokode)
  - Overordnet beskrivelse av hvordan alle delene av programmet henger sammen.
- Hvor mye dokumentasjon er nok?
  - Jo større og mer komplekst et program er, jo viktigere er god dokumentasjon.
  - God navngivning og ryddig programkode minsker behovet for dokumentasjon.
  - Dokumentasjon skal være nyttig. Unyttig dokumentasjon er meningsløst.

Hvilken av disse funksjonene er enklest å forstå?

```
/** Denne funksjonen kalkulerer en verdi ved å iterere over alle
    verdiene i en tabell. */
static double kalkulér(double[] tabell) {
    double verdi = 0;
    for (int indeks_til_tabell = 0; indeks_til_tabell < tabell.length;
            ++indeks_til_tabell)
        verdi += tabell[indeks_til_tabell]; // Legg til verdi
    return verdi/tabell.length;
}

static double gjennomsnitt(double[] verdier) {
    double sum = 0;
    int antall = verdier.length;
    for (int i = 0; i < antall; ++i)
        sum += verdier[i];

    return sum/antall;
}
```

Javadoc kommentarer

- Genererer dokumentasjon fra kildekoden.
- Javadoc helper med å dokumentere:
  - Klasser
  - Felt
  - Metoder
- Spesielle Javadoc-kommentarer blir supplert informasjonen som `javadoc`-verktøyet finner ved å analysere programkoden.

```
/** Dette er en javadoc kommentar. */
```

- `javadoc`-verktøyet gjenkjenner viss *merker* (eng. *markup tags*) i Javadoc-kommentarene.

Mønster / Merker	Mening
`@param` <*parameternavn*> <*beskrivelse*>	Beskrivelse angår en formell parameter med det angitte parameternavnet.
`@return` <*beskrivelse*>	Beskrivelse angår returverdien fra metoden

Eksempel: CD-samling

- Bruk av flere klasser.
- Bruk av konstruktører.
- Opprettelse av objekter, inklusive tabeller.
- Meldinger mellom objekter.
- Oppsett for å lage en applikasjon.
- Skjule medlemmer ved hjelp av adgangsmodifikatorer.
- Bruk av `javadoc`.

```
/**
 * Administrasjon av CD samling.
 * Bruker objekt av klasser CD_Samling og CD.
 */
public class CD_admin {
    public static void main (String[] args) {
        CD_Samling musikk = new CD_Samling(4);
        musikk.utskrift();

        musikk.innsett_CD (new CD(115.50, 4));
        musikk.innsett_CD (new CD(200,12));
        musikk.innsett_CD (new CD(99.99, 5));

        musikk.utskrift();

        musikk.innsett_CD (new CD(150.99, 12));
        musikk.innsett_CD (new CD(123.50, 15));

        musikk.utskrift();
    }
}  // klasse CD_admin
```



```
/** Denne klassen representerer en samling av Cder. */
class CD_Samling {
    /** Indeks til siste CDen innsatt. */
    private int inn_CD_indeks;
    /** Tabell som inneholder CDer. */
    private CD[] CD_tabell;

    /** Oppretter en CD samling.
     *  @param maks_antall_CDer maks antall CDer som skal lagres i samlingen.
     */
    public CD_Samling (int maks_antall_CDer) {
        inn_CD_indeks = -1;
        CD_tabell = new CD[maks_antall_CDer];
    }
    /** Innsetter en CD.
     *  @param nyCD Cden som skal settes inn.
     */
    public void innsett_CD (CD nyCD) {
        if (!erFull()) {
            ++inn_CD_indeks;
            CD_tabell[inn_CD_indeks] = nyCD;
            System.out.println ("Ny CD innsatt.");
        } else
            System.out.println ("Ikke mer plass i samlingen.");
    }
```

```
    /** @return Antall CDer i samlingen. */
    public int antallCDer() { return (inn_CD_indeks +1); }

    /** @return Sann dersom samlingen er full. */
    public boolean erFull() { return antallCDer() >= CD_tabell.length; }

    /** @return Sann dersom samlingen er tom. */
    public boolean erTom() { return (antallCDer() <= 0); }

    /** Skriver opplysninger om samlingen. */
    public void utskrift() {
        System.out.println ("***************");
        System.out.println ("Antall CDer: " + antallCDer());
        System.out.printf ("Verdi av samlingen: Kr %.2f%n", total_verdi());
        System.out.printf ("Gjennomsnittspris pr. CD: Kr %.2f%n",
                           gjennomsnitt_verdi_pr_cd());
        System.out.println ("***************");
    }
```



```
    /** @return Total verdi av CDene. */
    public double total_verdi () {
        if (erTom()) return 0.0;
        double sum_verdi = 0.0;
        for (int i = 0; i <= inn_CD_indeks; i++)
            sum_verdi += CD_tabell[i].hentPris();
        return sum_verdi;
    }

    /** @return Gjennomsnittsverdi av CDene. */
    public double gjennomsnitt_verdi_pr_cd () {
        if (erTom()) return 0.0;
        return total_verdi() / antallCDer();
    }
}  // klasse CD_Samling
```

```
/** Denne klassen representerer en CD. */
class CD {

    /** Pris til CDen. */
    private double CD_pris;
    /** Antall spor på CDen. */
    private int antall_spor;

    /** Oppretter en CD.
     *  @param pris Angir pris på CDen.
     *  @param spor Angir antall spor på CDen.
     */
    public CD(double pris, int spor) {
        CD_pris = pris;
        antall_spor = spor;
    }

    /** @return Pris til CDen. */
    public double hentPris() { return CD_pris;}

    /** @return Antall spor på CDen. */
    public int hentAntallSpor() { return antall_spor;}
}  // klasse CD
```


Utskrift fra programmet:

```
***************
Antall CDer: 0
Verdi av samlingen: Kr 0,00
Gjennomsnittspris pr. CD: Kr 0,00
***************
Ny CD innsatt.
Ny CD innsatt.
Ny CD innsatt.
***************
Antall CDer: 3
Verdi av samlingen: Kr 415,49
Gjennomsnittspris pr. CD: Kr 138,50
***************
Ny CD innsatt.
Ikke mer plass i samlingen.
***************
Antall CDer: 4
Verdi av samlingen: Kr 566,48
Gjennomsnittspris pr. CD: Kr 141,62
***************
```

Javadoc

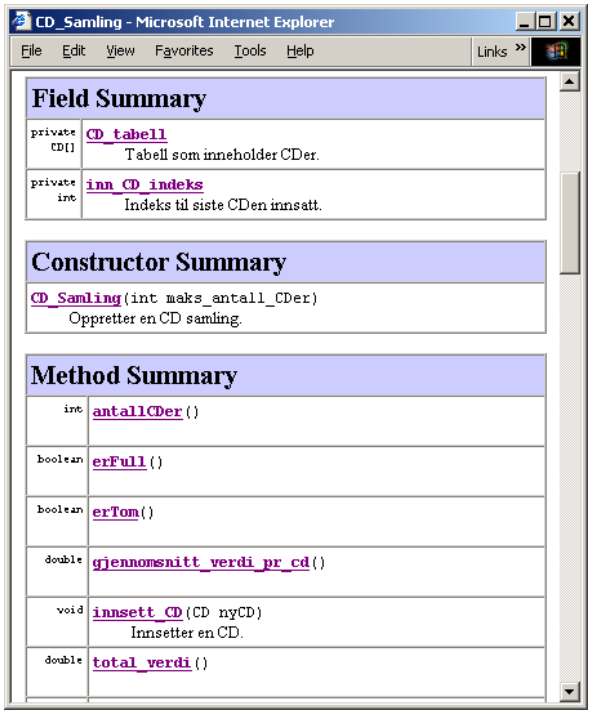
1. `javadoc` *<navn på Java-filer>*
2. `javadoc -private` *<navn på Java-filer>*

1. Genererer dokumentasjon for `public`, `package` og `protected` deklarasjoner (klasser, kontrakter, metoder, felt).
2. Genererer dokumentasjon for ALLE deklarasjoner (klasser, kontrakter, metoder, felt).

- Eksempel på bruk:

```
javadoc CD.java CD_Samling.java CD_admin.java
javadoc *.java
javadoc -private *.java
```


Generert dokumentasjon (`javadoc`)



Erfaring

- Evnen til å lage gode abstraksjoner, skrive godt strukturert programkode og god dokumentasjon kommer med erfaring.
- Den beste måten å få erfaring:
  - Skrive programkode.
  - Kompilere og finne feil.
  - Lese andres programkode.
  - Gjøre endringer i eksisterende programkode.
  - Vedlikeholde programkode over lengre tid.
- Erfaring hjelper deg å verdsette god programkode med god dokumentasjon
- Som alltid: *Øvelse gjør mester.*