Kapittel 7:
Mer om arv

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

Programmering med arv:
- En superklasse med flere subklasser
- Polymorfisme og dynamisk metodeoppslag
- Bruk av polymorfe referanser
- Konsekvenser av polymorfisme

Kontrakter i Java:
- Abstrakte metoder
- Kontrakttype
- Multippel arv for kontrakter
- Kontrakter og polymorfe referanser
- Super- og subkontrakter

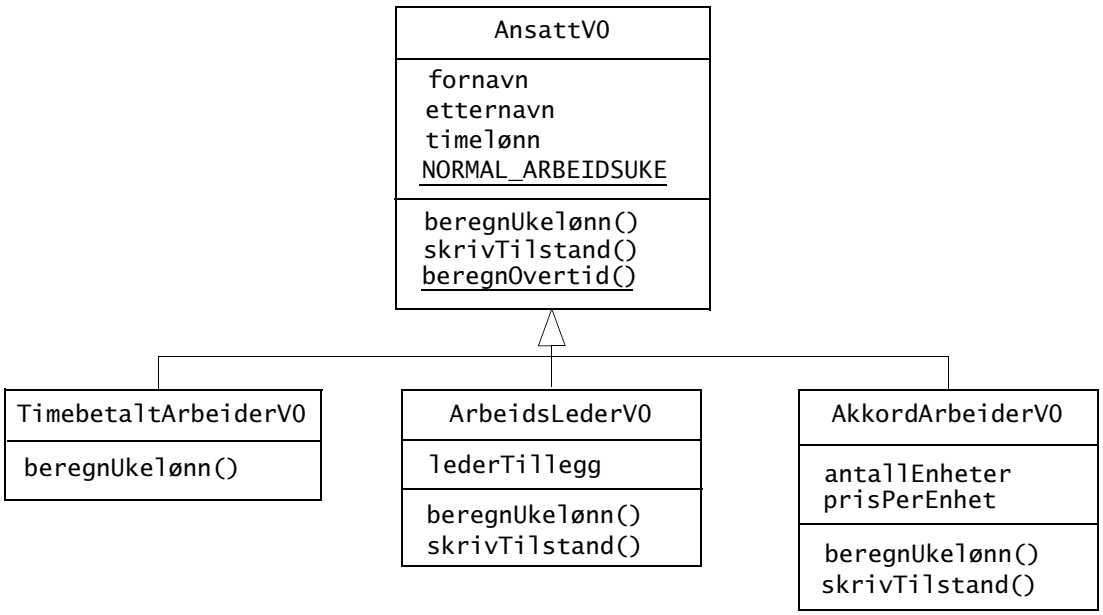
Abstrakte klasser:
- Instansiering og arv
- Metodeoverkjøring
- Konkrete klasser

Arv kontra aggregering

Substitusjonsprinsippet - Liskov

Programmering med arv

- En superklasse har ofte flere subklasser, som deklarerer nye eller overskygger felt, og/eller overkjører instansmetoder for å oppfylle subklassens abstraksjon.



Polymorfe referanser

- Hvilket objekt en referanse skal betegne under utføring kan ikke alltid avgjøres *under kompilering*.

```
...
AnsattV0 enAnsatt;
Scanner tastatur = new Scanner(System.in);
System.out.println("Velg 1=ansatt, 2=timebetalt, 3=leder, 4=akkord:");
int valg = tastatur.nextInt();
if (valg == 1) enAnsatt = new AnsattV0("Nils", "Nilsen", 150.0);
else if (valg == 2) enAnsatt = new TimebetaltArbeiderV0("Jon", "Ås", 0);
else if (valg == 3) enAnsatt = new ArbeidsLederV0("Ottar", "Boss", 350, 600);
else if (valg == 4) enAnsatt = new AkkordArbeiderV0("Per", "Berg");
else enAnsatt = new AnsattV0("Ole", "Olsen", 125.50);
// Hvilket objekt betegner enAnsatt ved utgangen av if-setningen?
...
```

- Kompilatoren kan ikke avgjøre hvilket objekt referansen `enAnsatt` vil betegne ved utgangen av `if`-setningen ovenfor.
- En *polymorf referanse* er en referanse som kan referere til objekter av forskjellige klasser til forskjellige tider.
  - En *referanse til en superklasse* er polymorf siden den kan referere til objekter av alle subklasser til superklassen under utføring.

Dynamisk metodeoppslag

- Når en metode blir påkalt på et objekt er det *klassen til objektet (dvs. aktuell type)* og *metodens signatur* som avgjør hvilken *metodedefinisjon* som blir utført.
  - *Typen til referansen (dvs. deklarert type)* som betegner objektet er *ikke* relevant i denne sammenhengen.

```
...
// Hvilket objekt betegner enAnsatt ved utgangen av if-setningen?
System.out.println("Ukelønn="
    + enAnsatt.beregnUkelønn() // Hvilken beregnUkelønn()-metode blir utført?
    );
enAnsatt.skrivTilstand();  // Og hvilken skrivTilstand()-metode blir utført her?
...
```

- *Dynamisk metodeoppslag* er prosessen som bestemmer hvilken *metodedefinisjon* metodesignaturen til kallet betegner under utføring, basert på klassen til objektet .
  - Denne prosessen *kan medføre søking oppover i arvhierarkiet for å finne metodedefinisjon*.
  - For det første kallet overnfor utføres `beregnUkelønn()`-metoden i den klassen som objektet `enAnsatt` betegner tilhører.
  - For det andre kallet blir `skrivTilstand()`-metoden i `AnsattV0` utført, dersom objektet `enAnsatt` betegner tilhører `AnsattV0`- eller `TimebetaltArbeiderV0`-klassen. Eller utføres `skrivTilstand()` i en av de to andre subklassene, avhengig av om `enAnsatt` betegner en arbeidsleder eller en akkordarbeider.

Bruk av polymorfe referanser

```
class AnsattTabell { // Fra Program 7.4
  static AnsattV0[] ansattTab = {                              // (1)
      new AnsattV0("Ole", "Brumm", 325.0),
      new TimebetaltArbeiderV0("Kari","Bø", 325.0),
      new AkkordArbeiderV0("Lasse", "Liten",  45.50, 126),
      new ArbeidsLederV0("Ottar", "Boss", 400.0, 500.0),
  };

  static double[] ansattTimer = { 37.5, 25.0, 30.0, 45.0 };    // (2)

  public static void main(String[] args) {
    // Går gjennom alle ansatte og skriver utvalgte egenskaper
    for (int i=0; i<ansattTab.length; i++){
      System.out.printf("Ansatt nr. %d: %s %s har en ukelønn på %.2f kroner%n",
          i+1, ansattTab[i].fornavn,                           // (3)
          ansattTab[i].etternavn,                              // (4)
          ansattTab[i].beregnUkelønn(ansattTimer[i]));         // (5)
    }
  }
}
```

- Utskrift fra kjøring:

```
Ansatt nr. 1: Ole Brumm har en ukelønn på 12187,50 kroner
Ansatt nr. 2: Kari Bø har en ukelønn på 8125,00 kroner
Ansatt nr. 3: Lasse Liten har en ukelønn på 5733,00 kroner
Ansatt nr. 4: Ottar Boss har en ukelønn på 21500,00 kroner
```


Konsekvenser av polymorfisme

- Polymorfisme lar oss betegne ulike typer objekter med en *felles (polymorf) referanse*, så lenge disse objektene tilhører en av subklassene eller superklassen i et *arvhierarki*.
- Dynamisk metodeoppslag gjør at samme kall i koden kan resultere i at *ulike metodeimplementasjoner blir utført under kjøring*.
  - *Aktuell type* bestemmer hvilken metode som blir utført.
- Klienter kan behandle super- og subklasseobjekter på samme måte, og trenger ikke endres selv om nye subklasser blir lagt til.
  - Dermed blir utvikling og vedlikehold av klienter enklere.

Kontrakter i Java

- En *kontrakt* i Java definerer et sett av tjenester - men *uten å gi noen implementasjon*.
- Kontrakten inneholder et sett av *abstrakte metodedeklarasjoner.*
- En *abstrakt metode* består av metodens navn, parameterliste og returtype — men ingen implementasjon.
  - Vi sier at en slik metode er *abstrakt*.
- Eksempel:

```
interface IIdrettslagsMedlem {
    double beregnKontingent();    // Abstrakt metode
}
```

- Klasser som vil *implementere* en kontrakt må angi dette vha. nøkkelordet `implements`, f.eks.

```
class Student implements IIdrettslagsMedlem {
    double beregnKontingent() { return 100.0; }
    // ... andre metoder og felt
}
```

  - og gir en implementasjon av kontraktens abstrakte metoder.

Kontrakter i Java (forts.)

- En kontrakt er *ikke* en klasse - så vi kan ikke opprette objekter av en kontrakt.
- Men en kontrakt definerer en type - *kontrakttypen* - og vi kan deklarere referanser av denne typen, f.eks.

```
IIdrettslagsMedlem enStudent;
```

- En referanse av en kontrakttype kan betegne objekter av *alle* klasser som implementerer kontrakten:

```
class AnsattVO implements IIdrettslagsMedlem {
    ...
    double beregnKontingent() { ... }
}
...
IIdrettslagsMedlem etMedlem;
etMedlem = new AnsattVO("Ole", "Olsen", 125.0);
...
etMedlem = new Student("Jens", "Jensen");
```

- Med en kontrakttype-referanse kan vi derfor få utført alle tjenester som kontrakten definerer.

Kontrakter i Java (forts.)

- En kontrakt kan også brukes til å importere *delte konstanter*.

```
interface ViktigeKonstanter {
    int AVSLUTT = -1;
    int FORTSETT = 1;
}

class ViktigKlasse implements ViktigeKonstanter {
    public int svar() {
        if (ferdig())
            return AVSLUTT; // Kontraktnavn ikke nødvendig pga implements.
        else
            return FORTSETT;
    }
    boolean ferdig() { ... }
}
```

- Dersom klassen `ViktigKlasse` ikke hadde deklarert at den implementerte kontrakten `ViktigeKonstanter`, måtte den ha brukt punktumnotasjon, f.eks. `ViktigeKonstanter.AVSLUTT`, for å få fatt i konstantene.

Kontrakter i Java (forts.)

- En klasse kan implementere flere kontrakter i Java. Dette kalles *multippel arv for kontrakter*.

```
class ArbeidsLederV0 extends AnsattV0
                     implements IFagforeningsMedlem, ITillitsValgt {
    ...
}
```

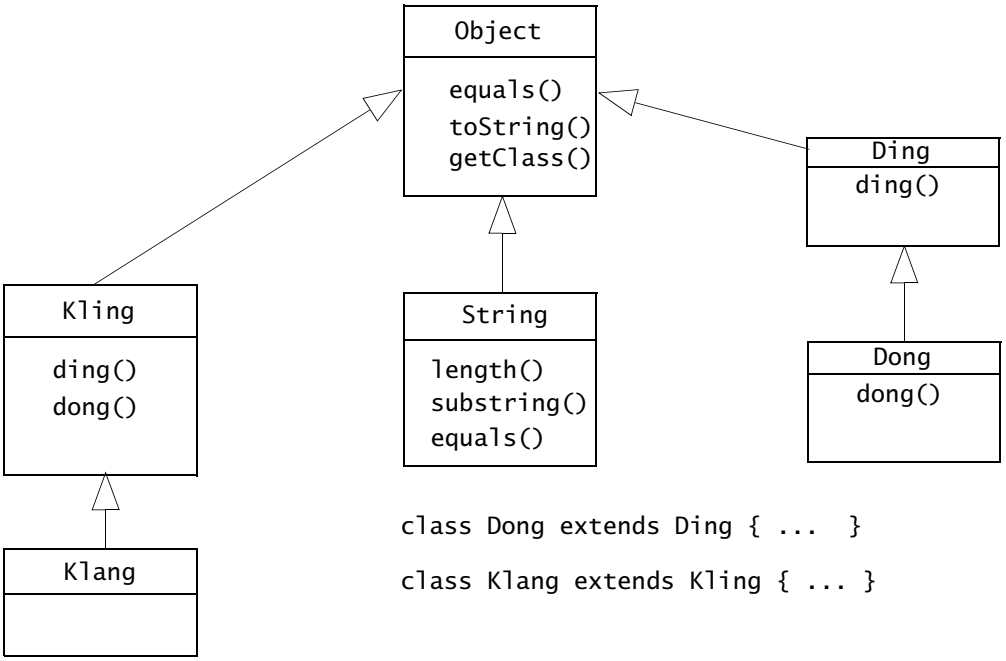
  – Klassen må da implementere alle abstrakte metoder fra samtlige kontrakter.

- En kontrakt `B` kan utvide en annen kontrakt `A`. Da kalles `A` en *superkontrakt* og `B` en *subkontrakt*.
- En klasse som implementerer `B` må implementere metodene fra *både* `A` og `B`.

```
interface ITikk { void tikk(); }
interface ITikkTakk extends ITikk { void takk(); }

class Klokke implements ITikkTakk {
    void tikk() { ... } // fra ITikk (superkontrakten)
    void takk() { ... } // fra ITikkTakk (subkontrakten)
}
```

Arvhierarki

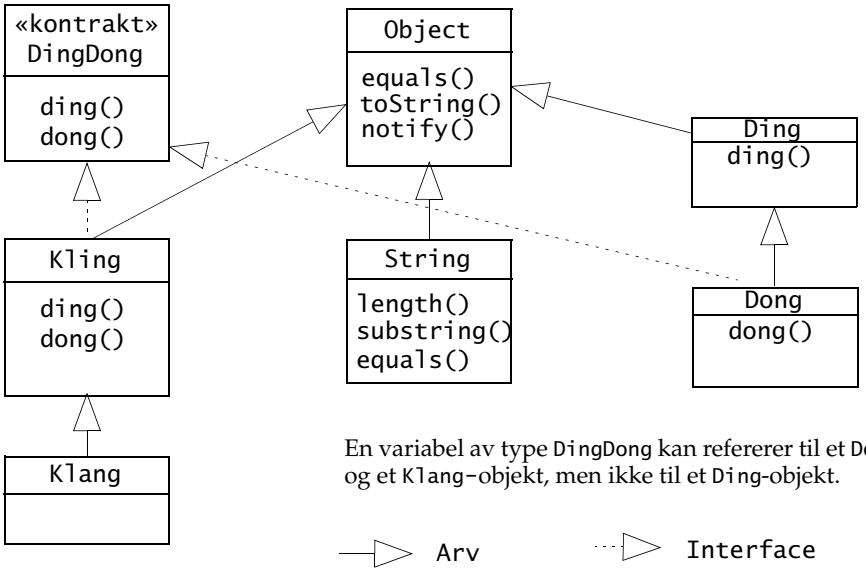



```
class Dong extends Ding { ...  }

class Klang extends Kling { ... }
```


Bruk av kontrakter for å håndtere objekter av flere typer

- Lag en *referanse type* som kan betegne ting som har både `dong()`- og `ding()`-metoder.



En variabel av type `DingDong` kan refererer til et `Dong`-, et `Kling`- og et `Klang`-objekt, men ikke til et `Ding`-objekt.

Bruk av kontrakter for å håndtere objekter av flere typer (forts.)

```
interface DingDong {
    public void ding();
    public void dong();
}

class Ding {
    public void ding() { System.out.println("ding fra Ding"); }
}

class Dong extends Ding implements DingDong {
    @Override public void dong() { System.out.println("dong fra Dong"); }
}

class Kling implements DingDong{
    @Override public void dong() { System.out.println("dong fra Kling"); }
    @Override public void ding() { System.out.println("ding fra Kling"); }
}

class Klang extends Kling {
    public void klang() { System.out.println("klang fra Klang"); }
}
```



```
public class TestInterface {
    public static void main(String[] args) {
        DingDong dingdonger[] = new DingDong[3];   // Oppretter tabell av referanser
        dingdonger[0] = new Dong();                // Initialiserer referansene
        dingdonger[1] = new Kling();
        dingdonger[2] = new Klang();
        for (int i = 0; i < dingdonger.length; i++)
            dingdonger[i].ding(); // Polymorfe referanser + dynamisk metodeoppslag
    }
}
```

- Utskrift fra kjøring:

```
ding fra Ding
ding fra Kling
ding fra Kling
```

Oppsummering av kontrakter i Java

- En klasse må eksplisitt deklarere at den implementerer en kontrakt.
- Klassen (eller dens subklasser) må implementere metoder for abstrakte metoder spesifisert i kontrakten.
- Dersom en klasse implementerer en kontrakt, implementerer også alle dens subklasser den automatisk p.g.a. arv.
  - Arv gjelder også for kontrakter: super- og subkontrakter.
- Kontrakter definerer en ny referanse type, og referanser av en kontrakttype er polymorfe.
- En klasse kan implementere *flere kontrakter*.
- En klasse som implementerer en subkontrakt må også implementere alle superkontraktens metoder.
- To varianter av arv:
  - *Design-arv*: Multippel-kontrakt arv der objekter kan tilhøre mange typer (vha. kontrakter i Java).
  - *Implementasjonsarv*: Enkel-implementasjonsarv der metode- og variabeloppslag er enklere (vha. subklasser i Java).

Nesten alle fordeler av generell multippel arv uten alle implementasjonsvanskeligheter.

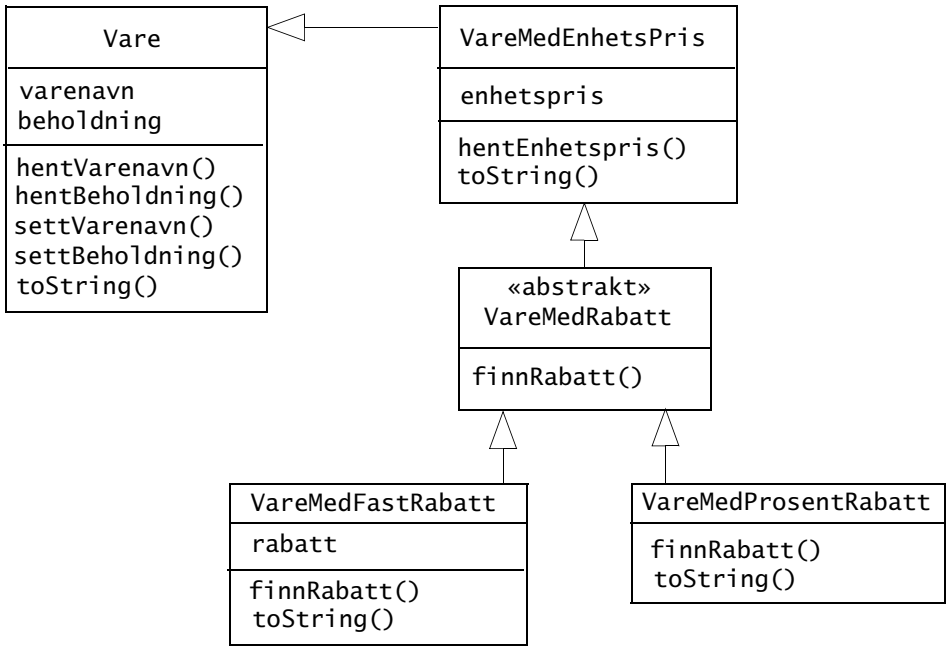
Abstrakte klasser

- En *abstrakt klasse* er en klasse som det ikke kan opprettes objekter av.
- Abstrakte klasser kan brukes som en superklasse, som gjennom arv tvinger alle subklasser til å overholde en felles kontrakt.
  - En abstrakt klasse *kan* implementere deler (eller hele) kontrakten. En Java-kontrakt derimot implementerer ingenting av kontrakten.
- En subklasse til en abstrakt klasse må implementere de abstrakte metodene i superklassen. Gjør subklassen ikke det, må den deklareres abstrakt.
- Klienter kan bruke *referanser av den abstrakte klassen* til å betegne subklasseobjekter, og vha. polymorfisme få utført riktige instansmetoder.

- Eksempel: Vi ønsker å gi rabatt på varer. Klassen `VareMedRabatt` blir *abstrakt superklasse* for alle rabatterte varer. Klassene `Vare` og `VareMedEnhetspris` gjenbrukes.

- Eksempel: Bruk av *Template Method* designmønster - en abstrakt klasse implementerer et felles rammeverk for oppførsel; subklasser må implementere de deler som er spesifikke for dem.

Abstrakte klasser (forts.)



Abstrakte klasser (forts.)

```
public class Vare {    // ... som tidligere
}
public class VareMedEnhetspris extends Vare {    // ... som tidligere
}

public abstract class VareMedRabatt extends VareMedEnhetspris {
    VareMedRabatt(String varenavn, int beholdning, double pris) {
        super(varenavn, beholdning, pris); // kaller superklassens konstruktør
    }
    abstract double finnRabatt(double prisUtenRabatt, int antallBestilt);
}

public class VareMedFastRabatt extends VareMedRabatt {
    private double rabatt;      // fast rabatt i kroner
    VareMedFastRabatt(String varenavn, int beholdning, double pris, double rabatt) {
        super(varenavn, beholdning, pris); // kaller superklassens konstruktør
        this.rabatt = rabatt;
    }
    @Override
    public double finnRabatt(double prisUtenRabatt, int antallBestilt){return rabatt;}
    @Override
    public String toString() { // overkjører toString() fra VareMedEnhetspris-klassen
        return super.toString() + "\tRabatt: " + rabatt; }
}
```

Abstrakte klasser (forts.)

```
public class VareMedProsentRabatt extends VareMedRabatt {
    int rabattIntervall;   // antall varer som må selges for å få rabatt
    double prosentSats;    // prosentsats rabatten økes med per intervall
    double maksimalSats;   // maksimal rabattsats

    VareMedProsentRabatt(String varenavn, int beholdning, double pris,
                         int rabattIntervall, double prosentSats, double maksimalSats) {
        super(varenavn, beholdning, pris); // kaller superklassens konstruktør
        this.rabattIntervall = rabattIntervall;
        this.prosentSats = prosentSats;
        this.maksimalSats = maksimalSats;}
    @Override
    public double finnRabatt(double prisUtenRabatt, int antallBestilt) {
        int antallIntervaller = antallBestilt / rabattIntervall;
        double rabatt = antallIntervaller * prosentSats;
        if (rabatt > maksimalSats) rabatt = maksimalSats;
        return rabatt/100.0*prisUtenRabatt;
    }
    @Override
    public String toString() { // overkjører toString() fra VareMedEnhetspris-klassen
        return super.toString() + "\tRabatt-intervall: " + rabattIntervall
                + "\tProsentsats: " + prosentSats + "\tMaks.sats: " + maksimalSats;
    }
}
```



```
public class Utsalg {
    public static void main(String[] args) {
        VareMedEnhetspris[] vareLager = {
                             new VareMedEnhetspris("Coca cola 0.5l", 150, 20.0),
                             new VareMedFastRabatt("Pepsi 0.5l", 100, 20, 0.50),
                             new VareMedEnhetspris("Solo 0.5l", 50, 20.0),
                             new VareMedFastRabatt("Farris 0.5l", 50, 20, 1.50),
                             new VareMedProsentRabatt("Coca cola 0.33l", 500, 12,
                                                      10, 0.5, 10.0) };
        int[] bestilling = { 50, 50, 10, 20, 250 };
        double sum = 0;
        double rabatt = 0;
        double varepris, varerabatt;
        for (int i=0; i<vareLager.length; i++) {
            varepris = vareLager[i].hentEnhetspris();
            varerabatt = 0;
            if (vareLager[i] instanceof VareMedRabatt) {
                VareMedRabatt vare = (VareMedRabatt) vareLager[i];
                varerabatt = vare.finnRabatt(varepris, bestilling[i]);
                varepris -= varerabatt;
                varerabatt *= bestilling[i]; // for hele ordren
            }
            varepris *= bestilling[i]; // for hele ordren
            System.out.println("Pris for " + bestilling[i] + " stk. av varen '"
```

```
                                + vareLager[i].hentVarenavn() +"' er " + varepris + " kr.");
            sum += varepris;
            rabatt += varerabatt;
        }
        System.out.println("Total sum: " + sum + " kr.");
        System.out.println("Rabatt: " + rabatt + " kr er trukket fra.");
    }
}
```

- Utskrift fra kjøring:

```
Pris for 50 stk. av varen 'Coca cola 0.5l' er 1000.0 kr.
Pris for 50 stk. av varen 'Pepsi 0.5l' er 975.0 kr.
Pris for 10 stk. av varen 'Solo 0.5l' er 200.0 kr.
Pris for 20 stk. av varen 'Farris 0.5l' er 370.0 kr.
Pris for 250 stk. av varen 'Coca cola 0.33l' er 2700.0 kr.
Total sum: 5245.0 kr.
Rabatt: 355.00000000000006 kr er trukket fra.
```

  - Merk at punktum brukes som desimaltegn av `println()`-metoden.
  - Øvelse: formater utskriften med komma som desimaltegn vha. `printf()`.

Arv kontra aggregering

- Arv tilbyr gjenbruk av klasser ved at *nye klasser får tilgang til variabler og metoder fra superklassen* (og alle superklasser lengre opp i arvhierarkiet).
- Aggregering tilbyr gjenbruk av klasser ved at *nye klasser deklarerer sammensatte objekter* der objekter av eksisterende klasser inngår som *delobjekter*.

- Arv brukes når:
  - den nye klassen er en naturlig spesialisering av en eksisterende klasse.
  - den eksisterende klassen kan utvides (dvs. ikke er en endelig klasse).
- Aggregering brukes når:
  - den nye klassen representerer sammensatte objekter, og det allerede er deklarert klasser for delobjektene.
  - en eksisterende klasse ikke kan utvides, derom den er endelig.

Merk at aggregering tilbys som standard i Java, mens for arv må vi bruke nøkkelordet `extends`.

Substitusjonsprinsippet - Liskov

- En referansetype kan være:
  - et klassenavn
  - et kontraktnavn
  - en tabelltype
- Subtyper og supertyper:
  - I arvhierarkiet, en referansetype kan ha en eller flere *supertyper*. Referansetypen er en *subtype* til hver av dens supertyper.
- Substitusjonsprinsippet:

  Subtypers atferd er i samsvar med spesifikasjon av deres supertyper, d.v.s. at et subtype-objekt kan substitueres for et supertype-objekt.