Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 1 – OOP och UML, repetition

Designmönster med C++

Syfte:	UML, repetera principer och begrepp för objektorienterad programmering.
Att läsa:	Design Patterns Explained kapitel 1 – 4 sid 3 – 71.

Lektion 1 – OOP och UML, repetition

Bokens undertitel är "A New Perspective on Software Design". Du ska ha detta i åtanke när du läser de inledande kapitlen. Det första designmönstret presenteras först på sidan 93 i boken. Det innebär inte att de inledande kapitlen är mindre viktiga. Tvärtom! De ger en mycket god grund för att du verkligen ska förstå styrkan i konceptet med objektorienterad design och programmering. Här introduceras den terminologi och bergreppsapparat som används senare i texten liksom de delar av UML som behövs för att tillgodogöra sig materialet. De inledande kapitlen ger troligen många aha-upplevelser även för den som anser sig vara erfaren inom området.

Några ord om terminologi. En funktion som definieras inom scopet av en klass kallas *medlemsfunktion* i C++ *(metod* i Java). Eftersom OOP och designmönster i grunden inte är knutet till något speciellt språk används på kursen det mer generella begreppet ***operation*** för en sådan funktion. På samma sätt används det språkneutrala begereppet ***attribut*** ofta för det som i C++ kallas *datamedlem* (kallas *instansvariabel* i Java).

Kapitel 1 –'The Object-Oriented paradigm'

Kapitel 1 jämför 'pre-OOP' programmeringsteknik baserad på funktionell uppdelning, med det objektorienterade paradigmet. Utgångspunkten är förmåga att hantera ändringar i krav och utvidgningar av program. Detta är ett centralt tema för boken: "Dealing with change". I det sammanhanget införs två viktiga begrepp:

Cohesion – hur starkt koden i en operation hänger samman, man talar ibland om kodens inre integritet.

Coupling – ett mått på hur starkt en operation är beroende av andra delar i ett program

En av fördelarna med det objektorienterade konceptet är att det kan underlätta att skriva kod med 'strong cohesion' och 'loose coupling' vilket ger program som är lätta att underhålla och att utvidga. Programmering baserad på traditionell funktionell uppdelning presterar betydligt sämre i dessa grenar! Som du senare ska se kommer alla designmönster som presenteras att underlätta att nå dessa mål.

En annan skillnad är ansvarsfördelningen. I ett funktionsbaserade program kommer ansvaret för det mesta som sker att ligga på main-funktionen medan en

central tanke inom OOP är att varje objekt så långt möjligt ska ansvara för sig själv. Vi byter alltså ett centralt ansvar mot en decentraliserad ansvarsfördelning.

Boken nämner tre olika perspektiv på utvecklingsprocessen.

- Det *konceptuella perspektivet* representerar koncepten inom det aktuella området. En konceptuell modell konstrueras utan några större hänsyn till den mjukvara som ska implementera den. Här besvaras frågor som ’Vad är jag ansvarig för?’ och ’Vad ska göras?’

- I *specifikationsperspektivet* betraktas de olika interfacen som ska användas i mjukvaran (men inte implementationerna). De resulterande interfacen besvarar frågan ’Hur ska jag användas?’

- *Implementationsperspektivet* handlar om kod. Detta är förmodligen det mest använda perspektivet på bekostnad av de övriga. Här besvaras frågan ’Hur ska jag kunna uppfylla mina uppgifter och åtaganden?’

Mer om detta kommer i senare lektioner.

Kapitel 2 – ‘The UML –The Unified Modeling Language’

Kapitlet ger en snabb översikt av de grundläggande delarna i UML. Här nedan följer kompletterande material som börjar med repetition av några grundläggande begrepp.

Klass:
En klass definierar en datatyp med tillhörande attribut och operationer. De publika medlemmarna utgör tillsammans klassens publika gränssnitt (public interface), dvs de medlemmar som kan accessas av klienter till klassen. Instanser av en klass kallas objekt.

Objekt:
Ett objekt är en unik instans av en bestämd klass/datatyp. Alla objekt av samma klass har samma uppsättning data (attribut), däremot har varje enskilt objekt sina *egna värden* på attributen. De operationer som kan utföras på ett objekt är de operationer som definierats för den klass som objektet tillhör.
I C++ kallas attributen datamedlemmar och operationerna utförs genom medlemsfunktioner.

I ett exekverande program kommunicerar olika objekt genom att skicka *meddelanden* (messages) till varandra. Objekt A skickar ett meddelande till

objekt B genom att anropa en medlemsfunktion för objekt B. Om meddelanden kan skickas mellan två objekt så existerar en relation mellan dessa objekt. En relation kan vara enkel- eller dubbelriktad.

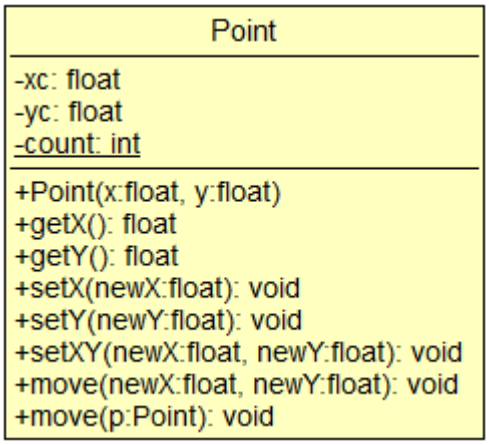
UML (Unified Modeling Language) är ett modelleringsspråk för att grafiskt beskriva objektorienterade system genom ett antal olika diagramtyper.

Den statiska strukturen hos en klass kan beskrivas i ett *klassdiagram* som utöver klassens namn bl.a. kan visa
- datamedlemmar (attribut)
- klassmedlemmar (static) visas understrukna
- medlemsfunktioner
- 'visibility'/access

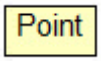
Exempel – klassen Point

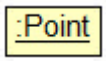
```
class Point {

public:      // public interface
   Point(float x, float y); // Constructor
   // getters
   float getX() const { return xc; }
   float getY() const { return yc; }
   // setters
   void setX(float newX) { xc = newX; }
   void setY(float newY)    { yc = newY; }
   void setXY(float,float);// set x and y
   void setXY(const Point &);
   void move(float, float);// move this point
   void move(float,float);
private:              // Private attributes
   float xc,yc;  // x-y coordinates
   static int count ; // Instance counter
};
```

Motsvarande UML klassdiagram:



Visibility/Access markeras med
- `-` för *private*
- `+` för *public*
- `#` för *protected*

Samma klassdiagram utan attribut och operationer kan skrivas 

En instans av klassen Point, dvs. ett objekt som existerar under exekveringen modelleras med 

Relationer

Det finns fem sorters relationer mellan klasser:

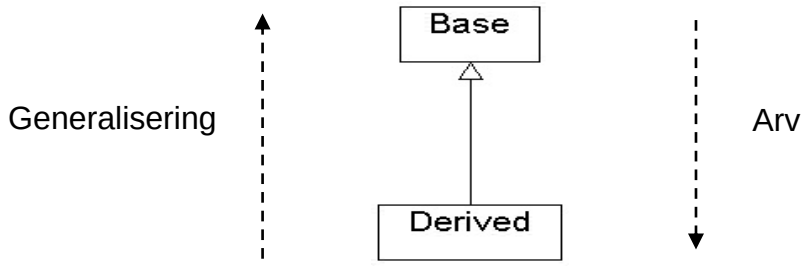
- En *generalisering* beskriver att en klass är en subklass till en annan.
- En *realisering* beskriver att en klassimplementerar ett gränssnitt (interface).
- En *association* beskriver att en klass har ett attribut av en annan klass eller omvänt.
- En *inkapsling* beskriver en situation där en klass är deklarerad inuti en annan för att göra den osynlig för andra klasser.
- Ett *beroende* beskriver en relation mellan två klasser av annat slag än de ovanstående som medför att den beroende klassen måste modifieras om den andra klassen förändras.

I ett klassdiagram visas relationer som linjer som förbinder klasser. Olika sorters associationer använder dekorationer som pilar, stereotyper och streckning. Att det finns en relation mellan två klasser syns i koden genom att klassnamnet för den ena klassen återfinns i den andra. Detta innebär att den senare klassen inte kan kompileras utan att den förstnämnda finns tillgänglig.

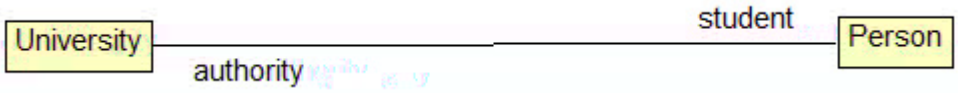
Generalisering (generalization)
Arv motsvaras i UML av relationen *generalisering*. Observera att begreppen arv och generalisering används för samma mekanism fastän de har olika utgångspunkter:

- Arv utgår från basklassen: en deriverad klass ärver basklassen.
- Generalisering utgår däremot från den deriverade klassen: basklassen är en generalisering av den deriverade klassen vilket också är innebörden av *substitutionsprincipen* som säger att en deriverad klass ska kunna ersätta sin basklass (men inte tvärtom).

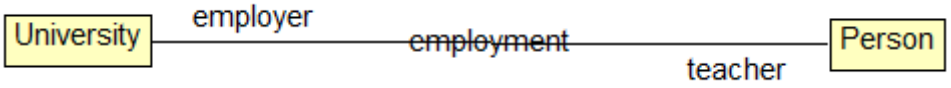
UML använder en ofylld triangel med en spets mot basklassen som symbol för generalization:



Association (association)
När en klass innehåller ett attribut (variabel/objekt) av en annan klass finns det en relation mellan klasserna som kallas association. Den mest allmänna formen av association ritas med ett streck mellan klasserna. Vid associationens ändar kan man ange ”roller”, som ofta överensstämmer med attributnamn i programmet.

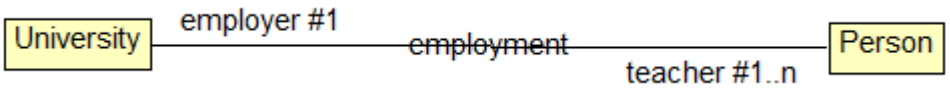


De två klasserna kan även vara associerade på andra sätt:



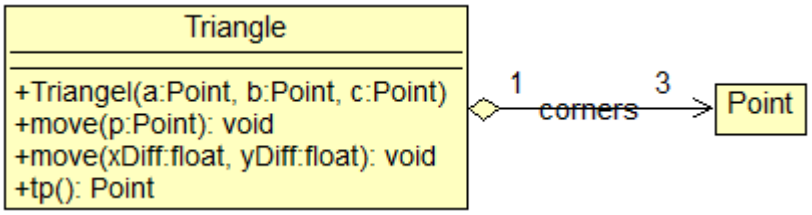
Kardinalitet, eller multiplicitet, anger antalet objekt av de olika typerna i en association.

Kardinalitet är heltal, som 1 eller 2, eller ett intervall, som 0..2. Man använder symbolen * för ett obegränsat antal, ensamt eller i intervall som 1..*.
Rollnamn och kardinalitet som används i en klass skrivs närmast den andra klassen och där är placeringen fri. Normalt sätter man ett namn på associationen ovanför eller på linjen som klargör relationen. Man kan också tillfoga en fylld triangel vid namnet som visar i vilken riktning namnet skall tolkas i förhållande till klasserna.
Nedanstående diagram visar en association, ”employment”, mellan University i rollen som arbetsgivare och Person i rollen som lärare. Innebörden är att universitetet kan ha godtyckligt många (men minst en lärare) och att en lärare bara kan vara anställd vid exakt ett universitet.

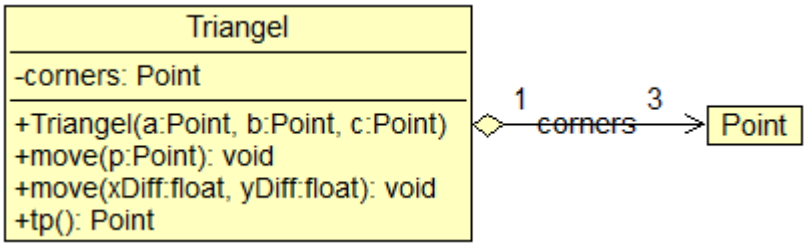


Eftersom en association innebär att en klass har ett attribut av den andra klassens typ så är det vanligt att inte visa denna datamedlem i klassrutan utan låta den utgöra namnet på associationen. Detta gäller inte primitiva typer som t.ex. int och float, dessa visas i stället oftast i klassrutan.

Detta innebär att klassdiagrammet



uttrycker samma sak som

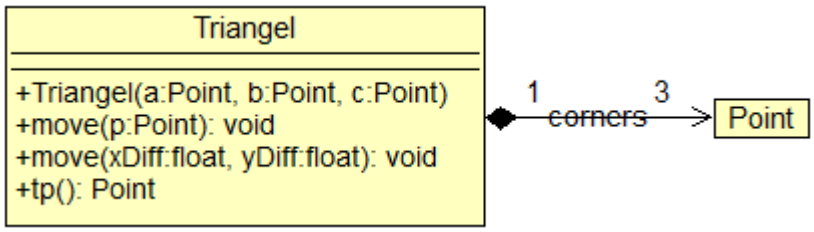


Ovanstående klassdiagram uttrycket ytterligare två saker:

- Den öppna pilen i riktning mot Point innebär att *navigering* endast är möjlig i den riktningen, dvs. Triangel kan skicka meddelanden till sina Point-objekt men att Point inte känner till Triangel. Om pilen saknas förutsätts att navigeringen är dubbelriktad.

- Den ofyllda romben i anslutning till Triangel anger att associationen är en *aggregation*, dvs en relation av typen **'uses-a'**. Triangel använder Point-objekt och accessar dessa via referenser/pekare som lagras i datamedlemmen corners. Objekten skapas utanför Triangel och skickas som argument till konstruktorn. Triangel 'äger' alltså inte dessa objekt. De kan i princip användas även av andra objekt.

En starkare form av koppling mellan Triangel och Point där Triangel skapar sina egna Point-objekt och där dessa endast existerar inom Triangelobjektets scope (räckviddsområde) och livslängd, kallas *composition*. De skapas och dör med det omgivande objektet. Detta uttrycks med en fylld romb:



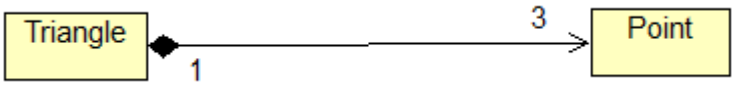
Denna typ av relation uttrycker alltså ett **'has-a'**-förhållande.

Exempel: TestTriangle (exTriangle.zip)

```
class Point {
public: // Public interface

    // Constructors
    Point();    // default constructor
    Point(float,float); // overloaded constructor

    // Getters & setters
    void setX(float newX) { xc = newX; } // set x-coord
    void setY(float newY) { yc = newY; }    // set y-coord
    Point setXY(float,float); // set x- and y-coord, returns old
Point
    Point setXY(const Point &); // set x and from argument
                                // returns old Point
    Point move(const Point &p); // move this point as indicated by p
                                // returns old Point
    float getX() const { return xc; } // get x-coord
    float getY() const { return yc; } // get y-coord

    // Facilitators
    void printOn(ostream &os=cout) const;  // print coordinates on
os

private:    // Private members
    float xc,yc;
};
```

```
class Triangel {
public:
    // Constructor
    Triangel(Point a, Point b, Point c);

   // Setters
    void move(Point);
    void move(float,float);

   // Getters
    Point tp() const;  // Center of mass, 'tyngdpunkt'

   // Facilitators
    void printOn(ostream &os=cout) const;
private:
    Point p1, p2, p3;

};
```

Relationen mellan Triangel och Point kan alltså uttryckas som



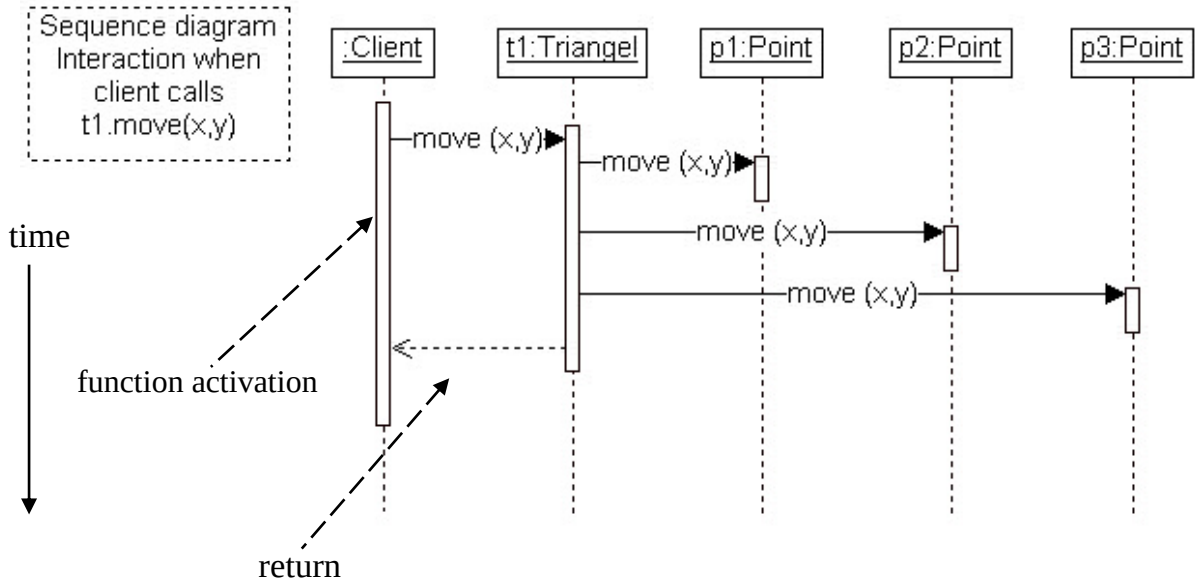
Triangel känner till och kan skicka meddelanden till Point men inte tvärtom.

Klassdiagram uttrycker endast struktur, inte hur objekt samarbetar under exekvering, dvs. dynamiska egenskaper. Detta uttrycks med hjälp av interaktionsdiagram. Ett mycket användbart sådant diagram är *sekvensdiagrammet*.

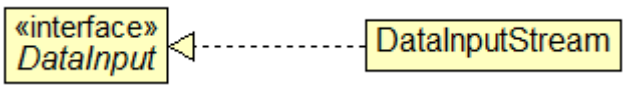
Sekvensdiagrammet nedan visar i tidsordning de anrop som görs när objektet Client anropar medlemsfunktionen move(Point):

```
void Triangel::move(Point p) {
    p1.move(p);
    p2.move(p);
    p3.move(p);
}
```

för Triangel-objektet t1:

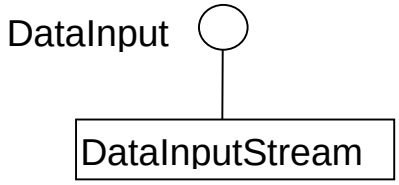


Realisering (realization)
En klass som implementerar ett gränssnitt (interface) ritas med en streckad pil med en ofylld triangel mot gränssnittet. Begreppet interface motsvaras i C++ av en abstrakt klass (minst en rent virtuell funktion).



I föregående figur är <<interface>> en sk. *stereotyp*. Stereotyper är tillägg till UMLs notation som man själv kan definiera för att förtydliga diagram. Användbara stereotyper är t.ex
<<create>>, <<destroy>> och <<interface>>.

En förenklad variant ersätter gränssnittsrutan med en liten cirkel (eng. lollipop) med namnet på gränssnittet vid sidan:



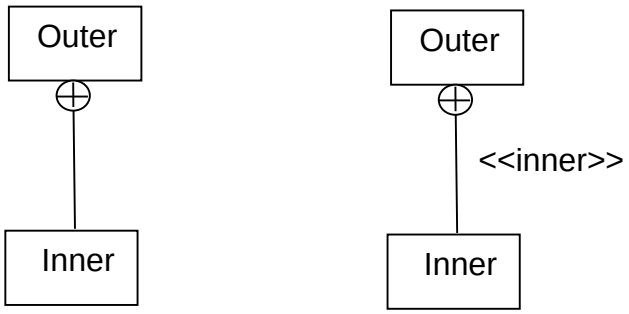
Inkapsling (encapsulation)
Inkapsling innebär (i det här sammanhanget) att man deklarerar en klass inuti en annan, detta kallas även att nästla klasser (nested classes). Anledningen kan vara att man vill ge objekt av den inre klassen direkt tillgång till alla attribut och metoder, även privata sådana, i den omgivande klassen samtidigt som det endast är den omgivande klassen som känner till och kan skapa objekt av den inre klassen. Klassen Inner blir alltså effektivt inkapslad i klassen Outer.

```
class Outer {
  private:
     int n;
     class Inner {
        ... n = 1; ...
     };
};
```

En variant är att man vill göra en klass osynlig utanför den omgivande klassen, men att den i övrigt inte skall ha några särskilda privilegier när det gäller access av icke-publika delar i den omgivande klassen. I detta fall skall den inre klassen deklareras som static (static nested class).

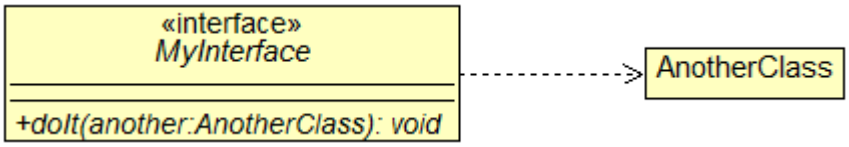
```
class Outer {
  private:
     static class Inner {
        ...
     };
};
```

Nästlade klasser används ofta för att representera den yttre klassens interna tillstånd och bör därför vara privata. I UML används följande notation för en sådan privatisering. Om man vill göra det tydligt vilken form av inkapsling det handlar om kan man introducera en egen stereotyp, t.ex. <<inner>>.

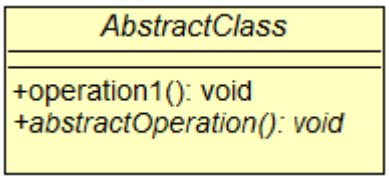


Beroende (dependency)
Om en klass har en metod med någon *parameter eller returvärde* av en annan klass utan att ha några attribut av denna klass ritar man relationen med en streckad beroendepil med pilen mot den andra klassen. I följande exempel har gränssnittet MyInterface en operation som tar ett argument av typen AnotherClass. Det innebär att MyInterface och alla klasser som implementerar detta måste ändras om det publika interfacet till AnotherClass ändras. MyInterface är alltså beroende av AnotherClass.

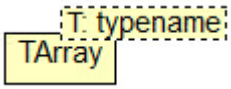
```
class MyInterface {
  public:
     void doIt(AnotherClass another)=0;
};
```



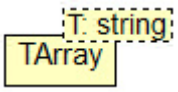
Abstrakta klasser och operationer
Namn på abstrakta klasser och metoder skrivs med *kursiv stil:*



Typparametriserade klasser
I C++ kan man definiera mallar (templates), klasser som tar andra klasser som typparametrar. När man instansierar en sådan klass anger man den aktuella elementtypen som argument. I UML beskrivs t.ex klassmallen TArray med den formella typparametern T som i nedanstående förenklade klassdiagram.



Mallen instansierad för string:



Kapitel 3 och 4_ - Ett problem och en (dålig) lösning

I kapitel 3 presenterar boken ett komplext problem som ska lösas med objektorienterad teknik. Problemet i sig är väl definierat och i kapitel 4 diskuteras en traditionell objektorienterard design för att lösa det.

Den givna lösningen baseras på arv. Från en basklass, Feature, deriveras klasser i flera led för varje tänkbart specialfall ska motsvaras av en egen klass.

Lösningen fungerar och uppfyller det viktiga kravet att den kan utvidgas med ytterligare specialfall utan att någon kod behöver skrivas om.

Men samtidigt lider den av svagheter som indikerar att det inte är den bästa lösningen:

- mycket av koden är redundant, dvs. samma kod förekommer på flera ställen
- koden är rörig
- ’tight copuling’ – om något av de grundläggande systemen ändras eller byts ut måste det mesta skrivas om
- ’weak cohesion’ – grundläggande metoder i systemet är spridda på flera klasser
- om de två CAD/CAM-systemen som lösningen ska understödja utökas med ett tredje kommer en ’kombinatorisk explosion’ att uppstå, 10 konkreta Feature-klasser blir i stället 15!

Systemet kommer att bli en mardröm att underhålla.

Exemplet visar att objektorienerad design i sig inte är någon garanti för att nå en bra och flexibel lösning. Senare i kursen återkommer problemet och då kommer också en betydligt bättre lösning baserad på designmönster att presenteras. Det är alltså viktigt att du förstår det presenterade problemet och svagheterna med den föreslagna lösningen för att du ska kunna tillgodogöra dig **Lösningen** senare i kursen.