Föreläsning 6

Top-Down Design
Parameteröverföring

Abstraktion

Ett datorprogram är en *modell* av verkligheten.

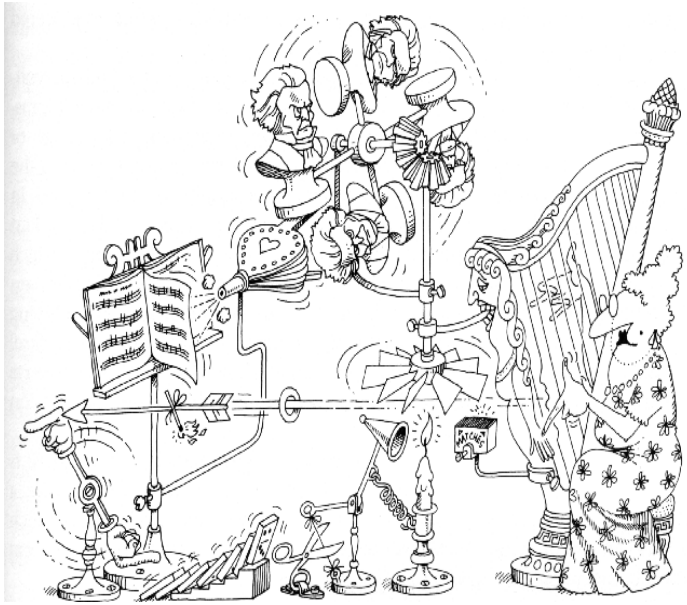
Ofta är det komplexa system som skall modelleras.

För att lyckas utveckla ett större program måste man arbeta efter en *metodik*.

En mycket viktig princip vid all problemlösning är att använda sig av *abstraktioner*.

En abstraktion innebär att man bortser från vissa omständigheter och detaljer i det vi betraktar, för att bättre kunna uppmärksamma andra för tillfället mer väsentliga aspekter.

Abstraktion är ett viktigt verktyg för att hantera komplexitet.



Top-Down Design

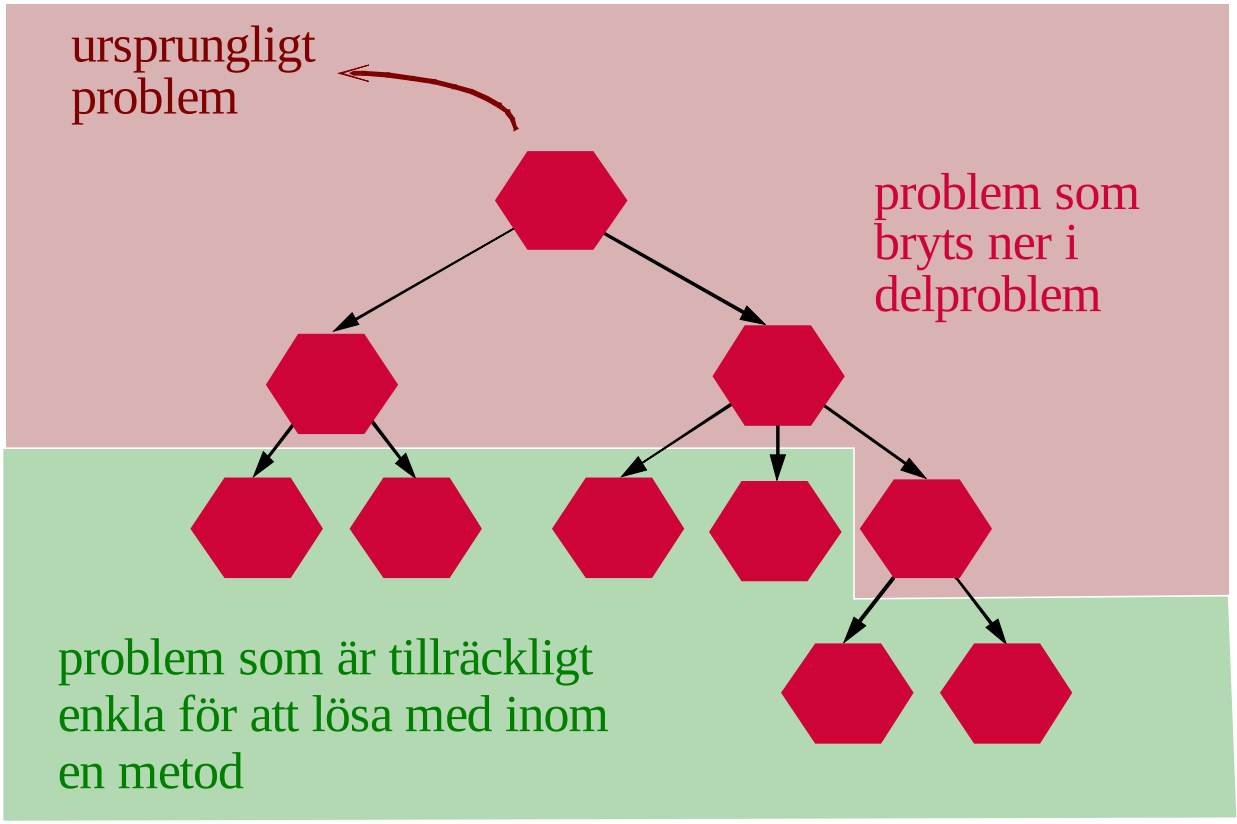
En problemlösningsmetodik som bygger på användning av abstraktioner är *top-down* design.

Top-down design innebär att vi betraktar det ursprungliga problemet på en hög abstraktionsnivå och bryter ner det ursprungliga problemet i ett antal delproblem.

Varje delproblem betraktas sedan som ett separat problem, varvid fler aspekter på problemet beaktas, dvs vi arbetar med problemet på en lägre abstraktionsnivå än vi gjorde med det ursprungliga problemet.

Om nödvändigt bryts delproblemen ner i mindre och mer detaljerade delproblem. Denna process upprepas till man har delproblem som är enkla att lösa. Top-down-design bygger på principen *divide-and-conquer*.

Top-Down Design



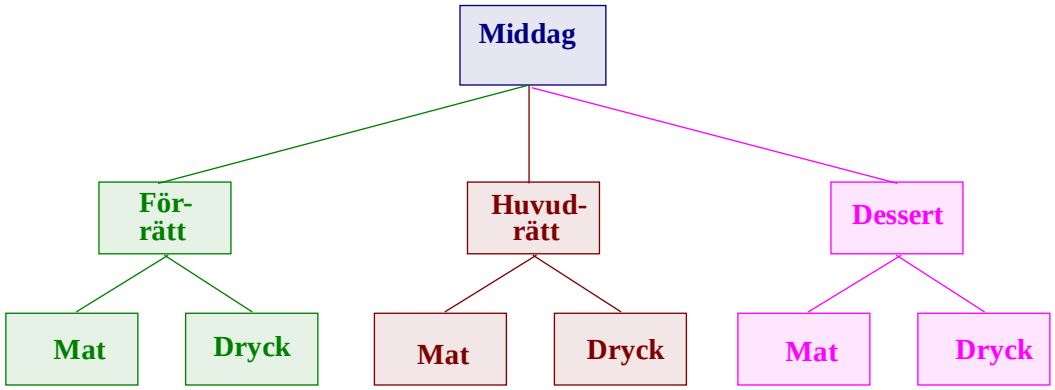
Med top-down design blir det möjligt att lösa det ursprungliga problemet steg för steg istället för att direkt göra en fullständig lösning.

Top-Down Design

Allteftersom ett problem bryts ner i mindre delproblem, betraktar man allt fler detaljer. Vi går således från en abstrakt nivå mot allt mer detaljerade nivåer. Denna process brukar kallas för *stegvis förfining*.

Exempel:

Att ordna en tre-rätters middag enligt "top-down design"



Top-Down Design

Vid utveckling av Javaprogram är klasser och metoder de abstraktions-mekanismer som används för att dölja detaljer och därmed öka *överblickbarhet* och *förståelse*.

Att utveckla ett Javaprogram med hjälp av top-down design innebär således att dela in programmet i lämpliga klasser och metoder, vilka i sin tur delas upp i nya klasser och metoder.

Man skall eftersträva att varje delproblem (= klass eller metod) handhar en *väl avgränsad uppgift* och att varje delproblem är så *oberoende* av de andra delproblemen som möjligt.

Exempel: Från laboration1

Skriv ett program som läser utgångshastigheten *v* (m/sek) och kastvinkeln α (i grader) och sedan beräknar banhöjden *h* och kastlängden *d* enligt nedanstående formler för kast utan luftmotstånd:

$$h = \frac{v^2 \cdot \sin^2 \alpha}{2 \cdot g} \qquad \text{och} \qquad d = \frac{v^2 \cdot \sin 2\alpha}{g}$$

där tyngdkraftsaccelerationen g = 9.81 m/sek^2.

Utkast till lösning:

1. Läs utgångshastigheten *v*.
2. Läs kastvinkeln α.
3. Beräkna banhöjden *h*.
4. Beräkna kastlängden *d*.
5. Skriv ut banhöjden *h* och kastlängden *d*.

Lösning 1:

Var och ett stegen i vår lösningsskiss är mer eller mindre triviala varför programmet kan skrivas som ett enda huvudprogram (som ni troligen gjorde på laboration 1 eftersom ni då inte visste bättre):

```
// Programmet läser in utgångshastighet och kastvinkel, samt skriver ut banhöjd och kastlängd
import javax.swing.*;
import java.text.*;
import java.util.*;
public class Kast1 {
   public static void main (String[] arg) {
      final double G = 9.81;
      String indata = JOptionPane.showInputDialog("Ange utgångshastighet och kastvinkel:");
      Scanner sc = new Scanner(indata);
      double v = sc.nextDouble();
      double alfa = sc.nextDouble();
      double h = (Math.pow(v, 2) * Math.pow(Math.sin(Math.toRadians(alfa)), 2))/(2*G);
      double d = (Math.pow(v, 2) * Math.sin(2*Math.toRadians(alfa)))/G;
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      JOptionPane.showMessageDialog(null, "Banhöjden är " + r.format(h)
                                          + "\noch kastlängden är " + r.format(d));
   } //main
} //Kast1
```

Lösning 2:

I lösningsskissen utgör dock varje steg ett delproblem och kan därmed implementeras som en metod! I implementationen nedan har vi valt att göra beräkningarna av banhöjd och kastlängd som egna metoder, de övriga stegen görs direkt i huvudprogrammet:

```
// Programmet läser in utgångshastighet och kastvinkel, samt skriver ut banhöjd och kastlängd
import javax.swing.*;
import java.text.*;
import java.util.*;
public class Kast2 {
   public static void main (String[] arg) {
      String indata = JOptionPane.showInputDialog("Ange utgångshastighet och kastvinkel:");
      Scanner sc = new Scanner(indata);
      double v = sc.nextDouble();
      double alfa = sc.nextDouble();
      double h = computeHeight(v, alfa);
      double d = computeDistance(v,alfa);
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      JOptionPane.showMessageDialog(null, "Banhöjden är " + r.format(h)
                                          + "\noch kastlängden är " + r.format(d));
   } //main
```

Lösning 2: fortsättning

```
    private static double computeDistance(double speed, double angel) {
        final double G = 9.81;
        return  (Math.pow(speed, 2) * Math.sin(2*Math.toRadians(angel)))/G;
    }//computeDistance

    private static double computeHeight(double speed, double angel) {
        final double G = 9.81;
        return (Math.pow(speed, 2) * Math.pow(Math.sin(Math.toRadians(angel)), 2))/ (2*G);
    } //computeHeight
} //Kast2
```

Kommentar:

Vi har deklarerat metoderna `computeDistance` och `ComputeHeight` **private** eftersom de är hjälpmetoder för att huvudprogrammet skall kunna göra sin uppgift.

Lösning 3:

I föregående implementation finns en lokal konstant `G` både i metoden `computeDistance` och metoden `computeHeight`. Denna är därför lämplig att göra global, dvs till en klasskonstant:

```
// Programmet läser in utgångshastighet och kastvinkel, samt skriver ut banhöjd och kastlängd
import javax.swing.*;
import java.text.*;
import java.util.*;
public class Kast3 {
   private static final double G = 9.81;

   public static void main (String[] arg) {
      //som tidigare
   } //main

   private static double computeDistance(double speed, double angel) {
      return  (Math.pow(speed, 2) * Math.sin(2*Math.toRadians(angel)))/G;
   }//computeDistance

   private static double computeHeight(double speed, double angel) {
      return (Math.pow(speed, 2) * Math.pow(Math.sin(Math.toRadians(angel)), 2))/ (2*G);
   } //computeHeight
} //Kast3
```

Lösning 4:

I föregående lösning har vi en klass som innehåller både ett huvudprogram och de privata klassmetoderna `computeDistance` och `computeHeight`.

Det är även möjligt (och lämpligt) att lägga metoderna `computeDistance` och `computeHeight` i en annan klass och än huvudprogrammet. Metoderna måste då göras publika. Det kan också vara lämpligt att deklarera konstanten `G` som publik, då denna kanske kan vara nyttig att ha tillgång till i andra tillämpningar.

```
public class Formler {
   public static final double G = 9.81;
   public static double computeDistance(double speed, double angel) {
      return  (Math.pow(speed, 2) * Math.sin(2*Math.toRadians(angel)))/G;
   }//computeDistance
   public static double computeHeight(double speed, double angel) {
      return (Math.pow(speed, 2) * Math.pow(Math.sin(Math.toRadians(angel)), 2))/ (2*G);
   } //computeHeight
} //Formler
```

Anm: Man skall alltid sträva efter att hålla sina klasser så små som möjligt. Detta underlättare både förståelsen för vad klassen gör, möjligheten till återanvändning av kod samt lokalisering av eventuella fel. *En klass skall göra en sak och göra denna sak bra.*

Lösning 4: fortsättning

```
// Programmet läser in utgångshastighet och kastvinkel, samt skriver ut banhöjd och kastlängd
import javax.swing.*;
import java.text.*;
import java.util.*;
public class Kast4 {
   public static void main (String[] arg) {
      String indata = JOptionPane.showInputDialog("Ange utgångshastighet och kastvinkel:");
      Scanner sc = new Scanner(indata);
      double v = sc.nextDouble();
      double alfa = sc.nextDouble();
      double h = Formler.computeHeight(v, alfa);
      double d = Formler.computeDistance(v,alfa);
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      JOptionPane.showMessageDialog(null, "Banhöjden är " + r.format(h)
                                          + "\noch kastlängden är " + r.format(d));
   } //main
} //Kast4
```

Bottom-Up Design

Ett alternativ till top-down design är *bottom-up design*.

Bottom-up design innebär att man startar med att utveckla små och *generellt användbara* programenheter och sedan bygger ihop dessa till allt större och kraftfullare enheter.

En viktig aspekt av objektorienterad programmering, som ligger i linje ned bottom-up design, är återanvändning. Återanvändning innebär en strävan att skapa klasser som är så generella att de kan användas i många program.

I Java finns ett *standardbibliotek* som innehåller ett stort antal sådana generella klasser. Standardbiblioteket kan således ses som en *"komponentlåda*" ur vilken man kan plocka komponenter till det programsystem man vill bygga.

Vid utveckling av Javaprogram kombinerar man vanligtvis top-down design och bottom-up design.

Uppbyggnaden av en metod

Metoder kan antingen vara klassmetoder eller instansmetoder.

Metoder kan antingen lämna ett returvärde eller inte lämna ett returvärde.

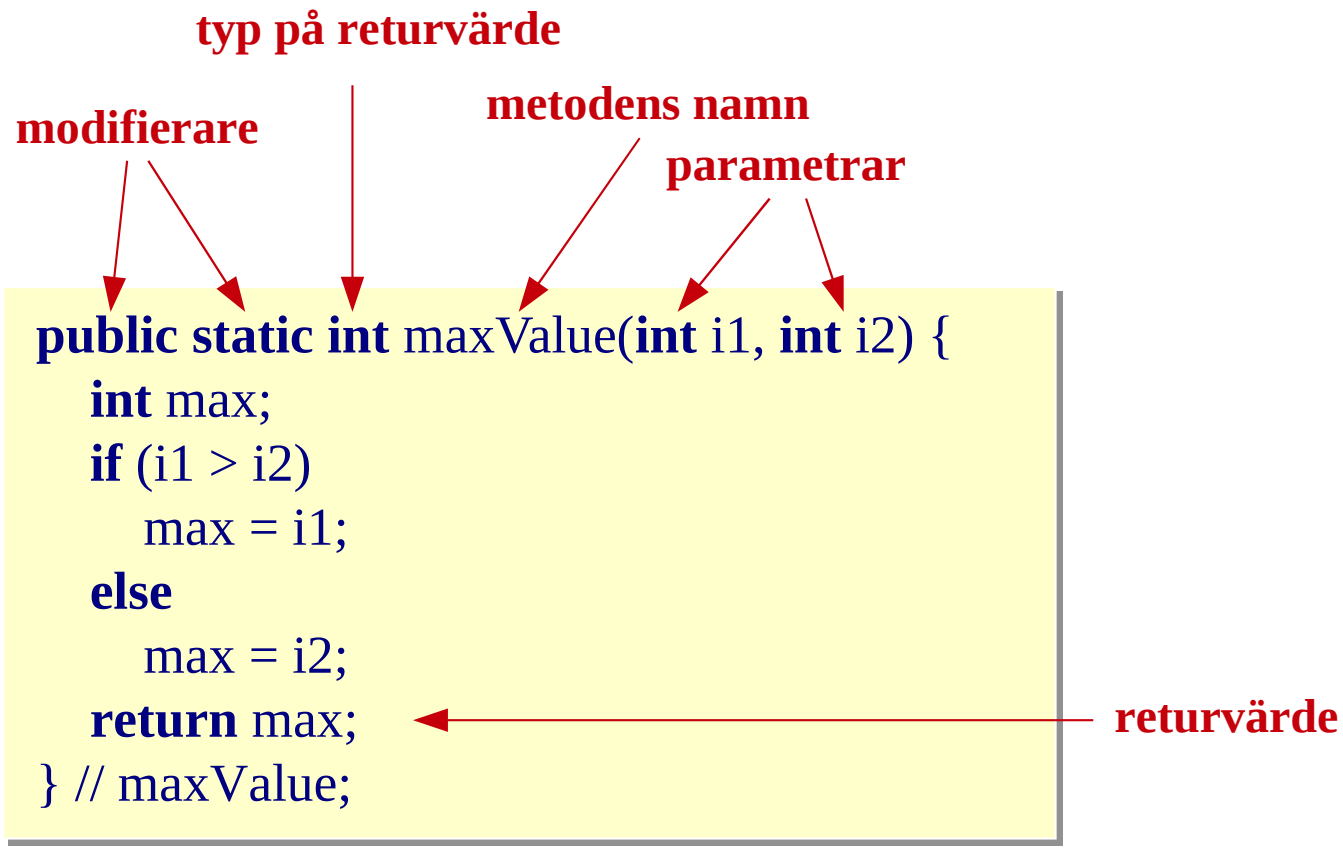
Metoder kan antingen vara **private**, **proceted** eller **public**.

```
//Utseende på metoder som lämnar returvärde
    modifierare typ namn(parametrar) {
        dataattribut och satser
        return uttryck;
    }
```

```
//Utseende på metoder som inte lämnar returvärde
    modifierare void namn(parametrar) {
        dataattribut och satser
    }
```

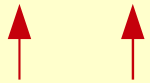
Uppbyggnaden av en metod

Exempel:



Formella och aktuella parametrar

```
import javax.swing.*;
public class Exempel {
   public static int  maxValue(int i1, int i2) {
      if (i1  > i2)
         return i1;
      else
         return i2;
   } //maxValue
   public static void main(String[] args) {
      String indata = JOptionPane.showInputDialog("Ge första talet:");
      int tal1 = Integer.parseInt(indata);
      indata = JOptionPane.showInputDialog("Ge andra talet:");
      int tal2 = Integer.parseInt(indata);
      indata = JOptionPane.showInputDialog("Ge tredje talet:");
      int tal3 = Integer.parseInt(indata);
      int big = maxValue(tal1, tal2);
      big = maxValue(big, tal3);
      JOptionPane.showMessageDialog(null, "Det största av talen " + tal1 + ", "
                                          + tal2 + " och " + tal3 + " är " + big);
   } // main
} //Exempel
```



formella parametrar

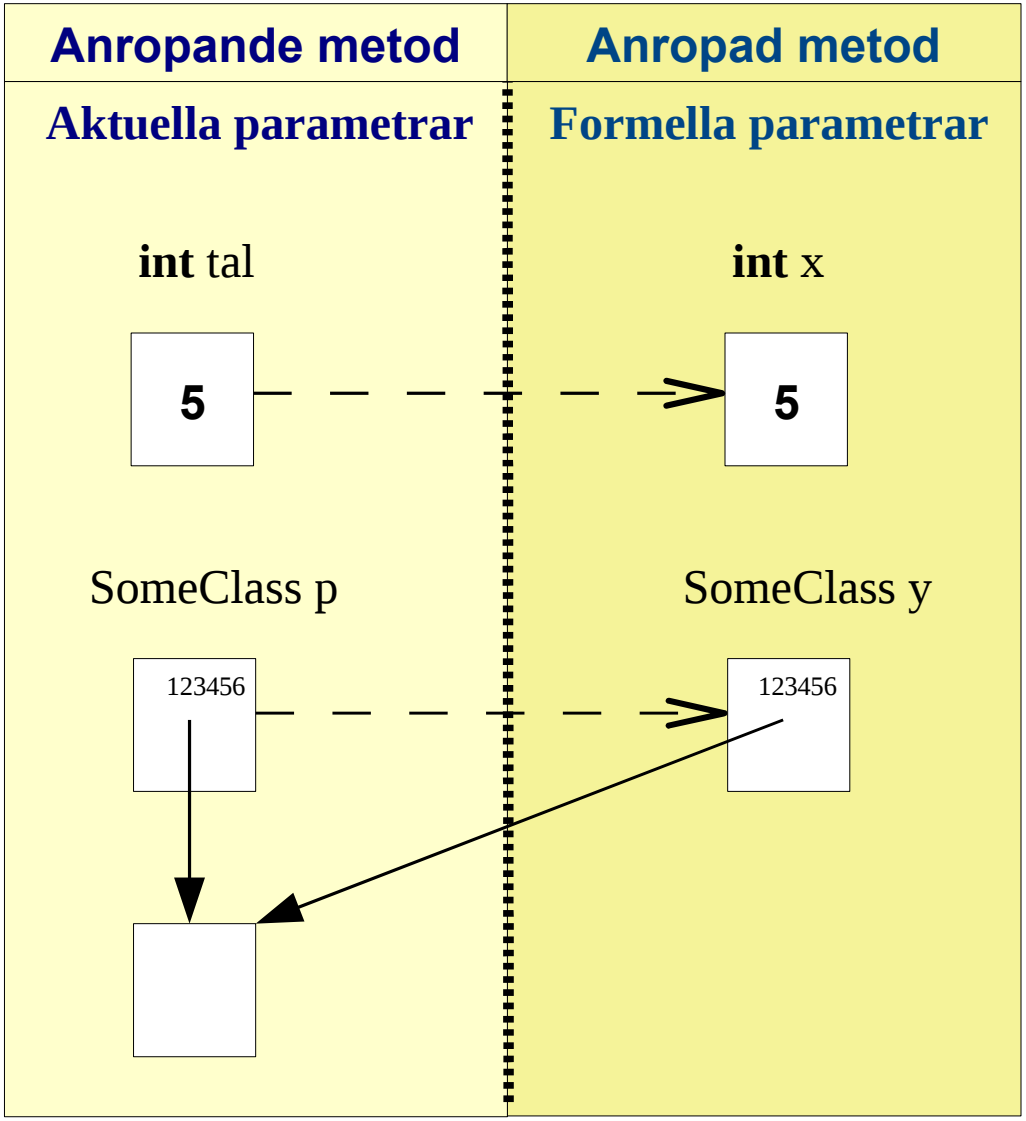
aktuella parametrar

Parameteröverföring

I Java sker alltid parameteröverföring via *värdeanrop*, vilket betyder att *värdet av den aktuella parametern kopieras över till den formella parametern*.

- När den aktuella parametern är en *primitiv typ* kommer därför den aktuella parametern och den formella parametern att ha access till *fysiskt åtskilda objekt*.
- När parametern är ett *objekt* (dvs en instans av en klass) är parametern en referensvariabel, varför den aktuella parametern och den formella parametern kommer att ha access till *samma fysiska objekt*.

Parameteröverföring



Värdet av den aktuella parametern tal kopieras till den formella parametern x.

tal och x är *åtskilda fysiska objekt*.

Värdet av den aktuella parametern p kopieras till den formella parametern y.

p och y kommer att referera till *samma fysiska objekt*.

Ännu mer om klasser och objekt

Vi har på de senaste föreläsningarna talat om klasser och objekt.

Av frågor som teknologer brukar ställa är min erfarenhet att begreppen klass och objekt upplevs som ganska svåra.

Därför kommer jag att återigen ta upp begreppen klass och objekt men göra detta från en något annan infallsvinkel än vad jag gjort tidigare.

Vi skall se att man kan lösa ett problem på flera olika sätt. Vi skall också se att de olika lösningarna kan struktureras på olika sätt och att lösningarna kan vara mer eller mindre objektorienterade.

Problemställning:

Sture Astoid är en inbiten astronom och vill ha din hjälp med att skriva ett program i vilket han på olika sätt kan simulera planeternas rörelser och förhållanden till varandra. Du skall hjälpa honom med att åstadkomma en grundstomme till ett sådant program.

Sture säger:

- att det skall vara möjligt ange planeternas storlek
- att en planet kan beskrivas som ett klot
- att det skall vara möjligt att ange planeternas position i rymden
- att från storleken på en planet beräkna dess volym
- att det skall vara möjligt att beräkna avståndet mellan två planeter.

Hur angriper du detta problem?

Jo, genom att ställa upp ett antal frågor och försöka besvara dessa.

- Hur beskrivs storleken på en planet?
- Hur beskrivs en position i rymden?
- Hur beräknas volymen av en planet?
- Hur beräknas avståndet mellan två planeter?

Analys:

Hur beskrivs storleken på en planet?

Med sin radie.

Hur beskrivs en position i rymden?

Med sina x-, y- och z-koordinater.

Hur beräknas volymen av en planet?

Med formeln $\frac{(4 \cdot \pi \cdot r^3)}{3}$, där r är planetens radie

Hur beräknas avståndet mellan två planeter?

Med formeln $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$, där ($x_1$, y_1, z_1) och (x_2, y_2, z_2) är positionerna för respektive position i planet.

Att *beskriva* en planet är att ange planetens tillstånd och tillstånden modelleras med hjälp av *instansvariabler* i klassen som definierar planeter.

Att *beräkna* en planets volym och att beräkna avståndet mellan två planeter är två väldefinierade delproblem som implementeras som två *metoder*.

Datarepresentation:

En planet beskrivs av sin radie samt sin position i rymden, dvs beskrivningen består av fyra komponenter

- radie
- x-koordinat
- y-koordinat
- z-koordinat

Samtliga dessa komponenter är reella tal, dvs representeras i ett Java med datatypen **double**.

```
double radie;
double x;
double y;
double z;
```

Vi kan således skapa en klass `Planet` för att avbilda planeter genom att låta ovanstående komponenter utgöra instansvariabler i klassen.

En första design:

I den första och mest primitiva (men inte bästa) implementationen av klassen `Planet` låter vi klassen enbart innehålla instansvariablerna. Därför måste instansvariablerna vara publika, annars skulle en användare varken kunna förändra tillståndet på en planet eller avläsa vilket tillstånd en planet befinner sig i.

```
public class Planet {
    public double radie;
    public double x;
    public double y;
    public double z;
}//Planet
```

Dålig lösning

Nu har vi möjlighet att skapa objekt av klassen `Planet`:

```
Planet p1 = new Planet();
Planet p2 = new Planet();
```

OBS! Om ingen konstruktor definieras i en klass, finns det automatiskt en parameterlös konstruktor att tillgå.

Implementation av metoderna

Innan vi kan implementera metoden att beräkna volymen av en planet och metoden att beräkna avståndet mellan två planeter måste vi

- bestämma metodernas *namn* och metodernas *parameterprofiler*

Namnen är enkelt, låt oss kalla metoderna

computeVolume
computeDistance

Både volymen och avståndet är reella tal, dvs metoderna returnerar ett värde av typen **double**.

Metoden som beräknar volymen av en planet måste veta radien av planeten och metoden som beräknar avståndet mellan två planeter måste veta planeternas position. Men både radien och positionen är komponenter i beskrivningen av planeten, varför det är naturligast att ge "hela planeten", dvs ett objektet av klassen `Planet`, som parameter istället för radie respektive position.

Implementation: Klassen `PlanetUtil`

Vi väljer att lägga metoderna

computeVolume
computeDistance

i en klass som döper till `PlanetUtil`. Metoderna skall kunna anropas av ett användarprogram, dvs de skall vara publika.

```
public class PlanetUtil {
   public static double computeVolume(Planet p) {
      return 4 * Math.PI*Math.pow(p.radie, 3) / 3;
   }//computeVolume
   public static double computeDistance(Planet p1, Planet p2) {
      return Math.sqrt(Math.pow(p1.x - p2.x,2) + Math.pow(p1.y - p2.y,2)
                  + Math.pow(p1.z - p2.z,2));
   }//computeDistance
}//PlanetUtil
```

Ett enkelt testprogram:

Låt oss nu skriva ett litet testprogram i vilket vi använder klasserna `Planet` och `PlanetUtil`.

```
import java.text.*;
public class TestPlanet {
    public static void main (String[] arg) {
        Planet p1 = new Planet();
        Planet p2 = new Planet();
        p1.radie = 1; p1.x = 100; p1.y = 100; p1.z = 100;
        p2.radie = 2; p2.x = 200; p2.y = 200; p2.z = 200;
        double volume = PlanetUtil.computeVolume(p1);
        double distance = PlanetUtil.computeDistance(p2, p1);
        NumberFormat r = NumberFormat.getInstance();
        r.setMaximumFractionDigits(2);
        System.out.println("Volym = " + r.format(volume));
        System.out.println("Avstånd = " + r.format(distance));
    }//main
}//TestPlanet
```

Programmet fungerar utmärkt, vi får de förväntade värdena på volymen och avståndet.

Men . . .

Vid en närmare eftertanke så beskriver ju metoderna `computeVolume` och `computeDistance` operationer som kan utföras på objekt av klassen `Planet`! Således bör dessa båda metoder finnas i klassen `Planet` och inte i en egen klass.

Version 2 av klassen `Planet`:

Dålig lösning

```
public class Planet {
   public double radie;
   public double x;
   public double y;
   public double z;
   public static double computeVolume(Planet p) {
      return 4* Math.PI*Math.pow(p.radie, 3) / 3;
   }//computeVolume
   public static double computeDistance(Planet p1, Planet p2) {
      return Math.sqrt(Math.pow(p1.x - p2.x,2) + Math.pow(p1.y - p2.y,2)
                       + Math.pow(p1.z - p2.z,2));
   }//computeDistance
} //Planet
```

Testprogram för Version 2:

Vårt testprogram får nu följande utseende:

```
import java.text.*;
public class TestPlanet {
   public static void main (String[] arg) {
      Planet p1 = new Planet();
      Planet p2 = new Planet();
      p1.radie = 1; p1.x = 100; p1.y = 100; p1.z = 100;
      p2.radie = 2; p2.x = 200; p2.y = 200; p2.z = 200;
      double volume = Planet.computeVolume(p1);
      double distance = Planet.computeDistance(p2, p1);
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      System.out.println("Volym = " + r.format(volume));
      System.out.println("Avstånd = " + r.format(distance));
   }//main
}//TestPlanet
```

Metoderna `computeVolume` och `computeDistance` är klassmetoder, vilket är orsaken till att de refereras med prefixet `Planet`.

Programmet fungerar utmärkt, vi får de förväntade värdena på volymen och avståndet.

Men . . .

Det är naturligare att låta metoderna `computeVolume` och `computeDistance` vara instansmetoder istället för klassmetoder! Det är ju alltid en *specifik planet* man vill beräkna volymen av respektive avståndet från.

Detta innebär att parameteruppsättningarna i metoderna blir annorlunda. Metoden `computeVolume` skall inte ha någon parameter, eftersom det är volymen på *objekt självt* som skall beräknas. Metoden `computeDistance` skall ha en parameter av klassen `Planet`, eftersom det är avståndet mellan objekt självt och ett annat objekt av klassen `Planet` som skall beräknas.

Version 3 av klassen `Planet`:

```
public class Planet {
    public double radie;
    public double x;
    public double y;
    public double z;

    public double computeVolume() {
        return 4* Math.PI*Math.pow(radie, 3) / 3;
    }//computeVolume

    public double computeDistance(Planet other) {
        return Math.sqrt(Math.pow(this.x - other.x, 2)
                + Math.pow(this.y - other.y, 2)
                + Math.pow(this.z - other.z, 2));
    }//computeDistance
} //Planet
```

Dålig lösning

Testprogram för Version 3:

Vårt testprogram får nu följande utseende:

```
import java.text.*;
public class TestPlanet {
    public static void main (String[] arg) {
        Planet p1 = new Planet();
        Planet p2 = new Planet();
        p1.radie = 1; p1.x = 100; p1.y = 100; p1.z = 100;
        p2.radie = 2; p2.x = 200; p2.y = 200; p2.z = 200;
        double volume = p1.computeVolume();
        double distance = p2.computeDistance(p1);
        NumberFormat r = NumberFormat.getInstance();
         r.setMaximumFractionDigits(2);
        System.out.println("Volym = " + r.format(volume));
        System.out.println("Avstånd = " + r.format(distance));
    }//main
}//TestPlanet
```

Programmet fungerar utmärkt, vi får de förväntade värdena på volymen och avståndet.

Men ..

Det är klumpigt att i programmet behöva skriva

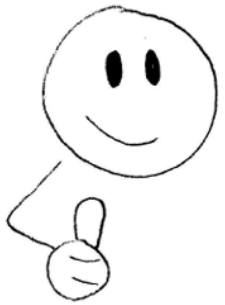
```
p1.radie = 1; p1.x = 100; p1.y = 100; p1.z = 100;
p2.radie = 2; p2.x = 200; p2.y = 200; p2.z = 200;
```

Så varför inte skaffa en konstruktor i vilken man kan initiera dessa värden när objekten skapas?

Har vi en sådan konstruktor behöver instansvariablerna inte längre vara publika utan skall göras privata.

Detta är en klar fördel, eftersom man då utanför klassen inte får direkt åtkomst till instansvariablerna. Instansvariablerna blir *dolda*, vilket *eliminerar risken för otillbörlig användning och manipulation av instansvariablerna*.

Version 4 av klassen `Planet`:



Bra lösning

```
public class Planet {
    private double radie;
    private double x;
    private double y;
    private double z;
    public Planet(double radie, double x, double y, double z) {
        this.radie = radie;
        this.x = x;
        this.y = y;
        this.z = z;
    }//konstruktor

    public double computeVolume() {
        return 4* Math.PI*Math.pow(radie, 3) / 3;
    }//computeVolume

    public double computeDistance(Planet other) {
        return Math.sqrt(Math.pow(this.x - other.x, 2)
                + Math.pow(this.y - other.y, 2)
                + Math.pow(this.z - other.z, 2));
    }//computeDistance
} //Planet
```

Komplettering av klassen `Planet`

I och med att instansvariablerna nu är dolda (har synlighetsmodifieraren **private**)

- kan inte tillståndet för en planet, dvs planetens position i rymden och dess storlek, avläsas i ett användarprogram
- kan int en planets position i rymden eller dess storlek förändras i ett användarprogram

Skall detta bli möjligt/tillåtet måste man utöka klassen `Planet` med ytterligare instansmetoder.

Man skall naturligtvis alltid överväga vilka tillstånd som skall vara tillåtna att förändra. Att en planets position förändras är uppenbart, men att en planets storlek förändras är kanske inte lika uppenbart. Vi tar beslutet att inte tillåta tillståndsförändringar på planeternas storlek.

Kompletteradande kod till Version 4 av klassen `Planet`:

```
public class Planet {
   . . .                  //samma instansvariabler och metoder som tidigare
   public Planet(double radie) {
       this.radie = radie;
   }//konstruktor
   public double getRadie() {
      return radie;
   }//getRadie
   public void setX(double x) {
      this.x = x;
   }//setX
   public double getX() {
     return x;
   }//getX
   public void setY(double x) {
       this.y = y;
   }//setY
   public double getY() {
      return y;
   }//getY
```

Bra lösning

```
   public void setZ(double z) {
      this.z = z;
   }//setZ
   public double getZ() {
      return z;
   }//getZ
   public void moveTo(double x, double y, double z) {
      this.x = x;
      this.y = y;
      this.z = z;
   }//setY
} //Planet
```

Testprogram för Version 4:

Vårt testprogram får nu följande utseende:

```
import java.text.*;
public class TestPlanet {
    public static void main (String[] arg) {
        Planet p1 = new Planet(1, 100, 100, 100);
        Planet p2 = new Planet(2, 200, 200, 200);
        double volume = p1.computeVolume();
        double distance = p2.computeDistance(p1);
        NumberFormat r = NumberFormat.getInstance();
        r.setMaximumFractionDigits(2);
        System.out.println("Volym = " + r.format(volume));
        System.out.println("Avstånd = " + r.format(distance));
    }//main
}//TestPlanet
```

Klassen `Point3D`

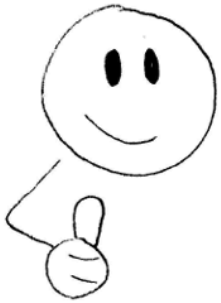
Om vi betraktar klassen `Planet` närmare finner vi att koordinaterna i rymden i sig utgör ett objekt och skulle därför kunna representeras med hjälp av en egen klass. Denna klass kan ha följande utseende:

```
public class Point3D {
   private double x;
   private double y;
   private double z;
   public Point3D() {}//konstruktor
   public Point3D(double x, double y, double z){
      this.x = x; this.y = y; this.z = z;
   }//konstruktor
   // metoden computeDistance beräknar avståndet mellan den aktuella punkten
   //och punkten other
   public double computeDistance(Point3D other) {
      return Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y - other.y, 2)
                       + Math.pow(this.z - other.z,2));
   }//computeDistance
   // Här skall finns fler metoder finnas såsom:
   // setX, getX, setY, getY, setZ och getZ
} //Point3D
```

Version 5: Klassen `Planet` med användning av klassen `Point3D`

Om vi utnyttjar klassen `Point3D` för att implementera klassen `Planet` får klassen endast två instansvariabler. Vidare kan vi utnyttja metoden `computeDistance` i klassen `Point3D` för att implementera metoden `computeDistance`.

```
public class Planet {
   private double radie;
   private Point3D position;
   public Planet(double radie) {
      position = new Point3D();
      this.radie = radie;
   }//konstruktor
   public Planet(double radie, Point3D position){
      this.radie = radie;
      this.position =  position;
   }//konstruktor
   public double getRadie() {
      return radie;
   }//getRadie
   public Point3D getPosition() {
      return position;
   }//getPosition
```



Bra lösning

```
   public void moveTo(Point3D position) {
      this.position = position;
   }//setPosition
   public double computeVolume() {
      return 4* Math.PI*Math.pow(radie, 3) / 3;
   }//computeVolume
   public double computeDistance(Planet other) {
      return position.computeDistance(other.position);
   }//computeDistance
} //Planet
```

Testprogram för Version 5:

Vårt testprogram får nu följande utseende:

```
import java.text.*;
public class TestPlanet {
   public static void main (String[] arg) {
      Point3D position1 = new Point3D(100, 100, 100);
      Point3D position2 = new Point3D(200, 200, 200);
      Planet p1 = new Planet(1, position1);
      Planet p2 = new Planet(2, position2);
      double volume = p1.computeVolume();
      double distance = p2.computeDistance(p1);
      NumberFormat r = NumberFormat.getInstance();
      r.setMaximumFractionDigits(2);
      System.out.println("Volym = " + r.format(volume));
      System.out.println("Avstånd = " + r.format(distance));
   }//main
}//TestPlanet
```

Uppgift

Sture Astoid är nöjd med klassen `Planet` som vi åstadkommit, men återkommer och säger att han också vill kunna beräkna gravitationen mellan två planeter.

Eftersom du just läser en fysikkurs och vet att gravitationen `F` mellan två kroppar, med massorna m_1 respektive m_2, som befinner sig på ett avstånd d från varandra ges av formeln

$$F = \frac{(G \cdot m_1 \cdot m_2)}{d^2}$$

där den universella gravitationskonstanten G är $6.670 \cdot 10^{-11}$ Nm^2/kg^2, åtar du dej uppdraget.

Hur kommer den nya versionen av klassen `Planet` att se ut?