Metoder och top-down design

Föreläsning 4

TDA540 - Objektorienterad Programmering

CHALMERS

Meddelande

- Bara några dagar kvar till laboration 1 deadlinen
- Läs textboken, läsanvisningar finns på hemsidan
- Kom ihåg flödet: läsa, föreläsning, öva/lab
- Finns extra instuderingsfrågor på hemsidan

Sammanfattning

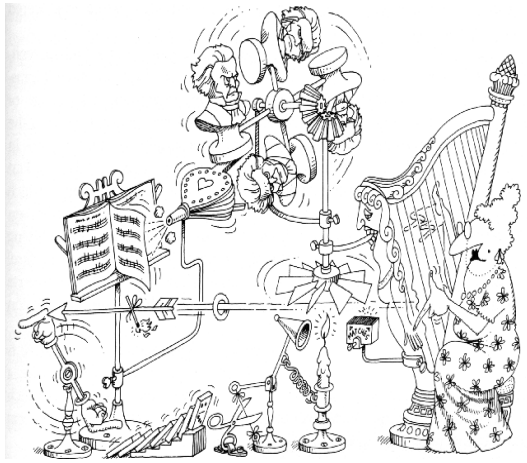
- Iteration: `while`-, `do`- och `for`-sats
- variablers räckvidd (scope)
- `double`: akta begränsad storlek och avrundningsfel

Programmering = modellering

Ett datorprogram är en *modell* av en verklig eller tänkt värld. Ofta är det komplexa system som skall modelleras

I objektorienterad programmering består denna värld av ett antal objekt som tillsammans löser den givna uppgiften.

- De enskilda objekten har specifika ansvarsområden.
- Objekten samarbetar genom att kommunicera med varandra via meddelanden.
- Ett meddelande till ett objekt är en begäran från ett annat objekt att få något utfört.



Att göra en bra modell av verkligheten, och därmed möjliggöra en bra design av programmet, är en utmaning.

Abstraktion

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 det viktigaste verktyget vi har för att hantera komplexitet och för att finna gemensamma drag hos problem och hitta generella lösningar.

Betraktas alla detaljer *ser man inte skogen för alla träden* och två problem kan synas helt olika, medan de på en hög abstraktionsnivå är identiska.

Viktigaste slide av hela kursen!

Top-Down Design

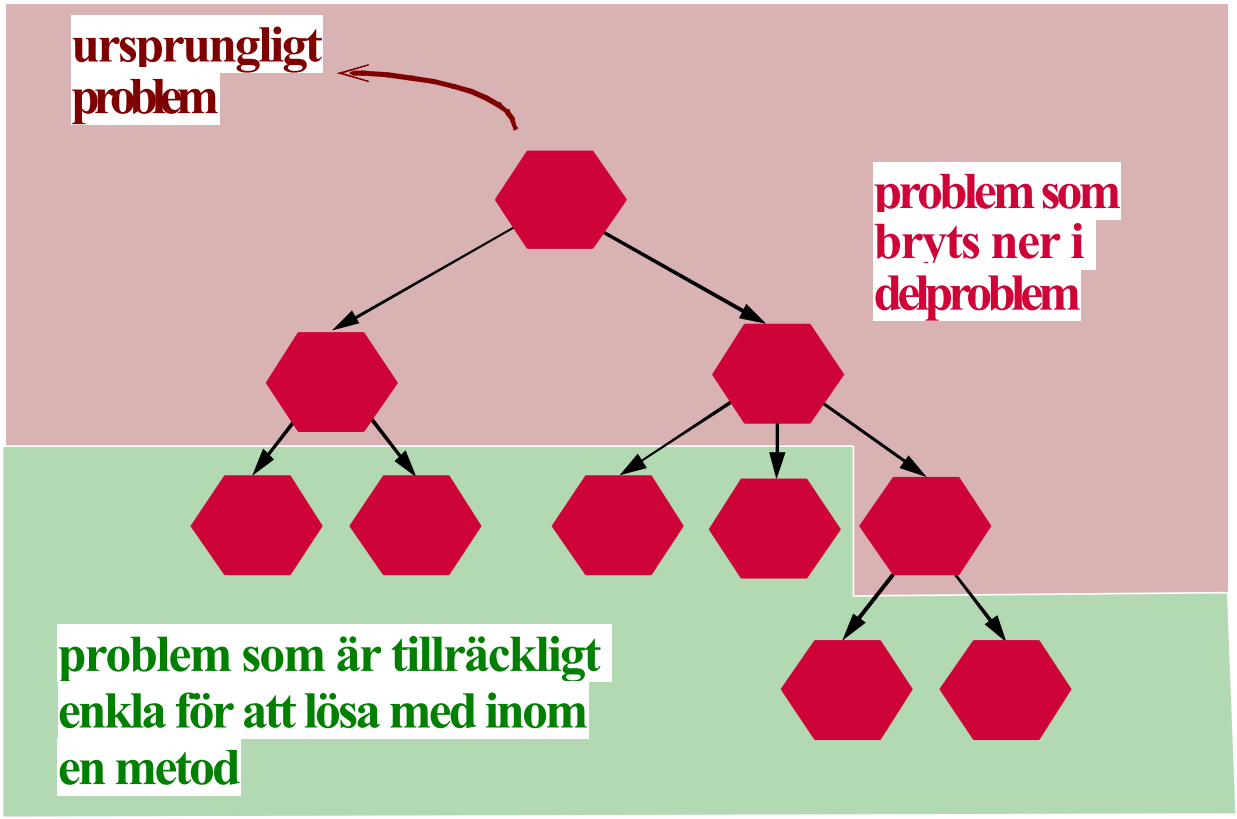
En problemlösningsmetodik som bygger på användning av abstraktioner är *top-down design*.

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 överblicka och lösa. Top-down-design bygger på principen *divide-and-conquer*.

Top-Down Design



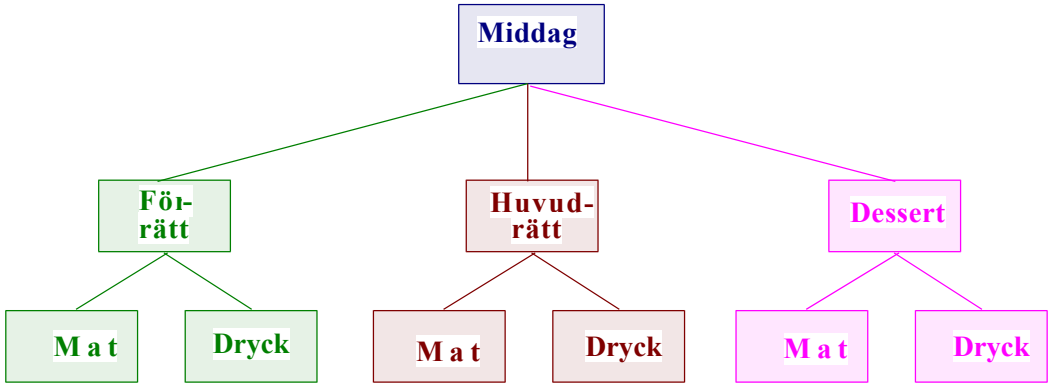
Med top-down design blir det möjligt att lösa det ursprungliga problemet steg för steg istället för att direkt göra en fullständig lösning.

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"



Modulär Design

Vid utveckling av Javaprogram är klasser och metoder (tillsammans med paket) de *abstraktionsmekanismer* 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 en *modulär design* där varje delproblem (= klass eller metod) handhar en *väl avgränsad uppgift* och att varje delproblem är så *oberoende* av de andra delproblemen som möjligt.

Modulär design

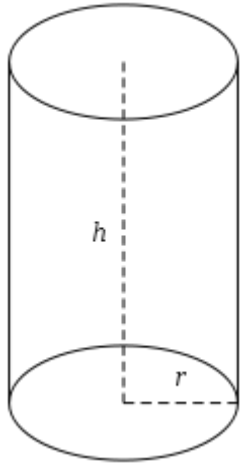
En *välgjord* modulär design innebär att programsystemet är uppdelat i *tydligt identifierbara abstraktioner*. Fördelarna med ett sådant system är:

- det går lätt att utvidga
- komponenterna går att återanvända
- komponenterna har en tydlig uppdelning av ansvar
- komplexiteten reduceras
- komponenterna går att byta ut
- underlättar testning
- tillåter parallell utveckling.

*Designa programsystemet runt **stabila abstraktioner** och **utbytbara komponenter** för att möjliggöra små och stegvisa förändringar.*

Exempel:

Skriv ett program som läser in radien och höjden av en cylinder, samt beräknar och skriver ut cylinderns area och volym.



Arean A och volymen V av en cylinder fås av följande formler:

$$A = 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2 \text{ och } V = \pi \cdot r^2 \cdot h$$

där r är radien och h är höjden av cylindern.

Utkast till lösning:

1. Läs cylinderns radie r.
2. Läs cylinders höjd h.
3. Beräkna cylinderns area A mha formeln $A = 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2$.
4. Beräkna cylinderns volym A mha formeln $V = \pi \cdot r^2 \cdot h$.
5. Skriv ut cylinderns area A och volym V.

Lösning 1:

Var och ett stegen i vår lösningsskiss är mer eller mindre triviala varför programmet kan skrivas som ett enda huvudprogram:

```
import javax.swing.*;
import java.util.*;
public class Cylinder {
  public static void main (String[] arg) {
    boolean done = false;
    while (!done) {
      String input = JOptionPane.showInputDialog("Ange radie och höjd:");
      if (input != null) {
        Scanner sc = new Scanner(input);
        double radius = sc.nextDouble();
        double height = sc.nextDouble();
        double area   = 2 * Math.PI * radius * height +
                        2 * Math.PI * Math.pow(radius, 2);
        double volume = Math.PI * Math.pow(radius, 2) * height;
        JOptionPane.showMessageDialog(null, "Arean av cylindern är "+ area +
                                            "\nVolymen av cylindern är " +
                                            volume);
        } else {
          done = true;
        }
    }
  }
}
```

Lösning 2:

I lösningsskissen utgör beräkningen av arean respektive beräkningen av volymen var sitt delproblem och kan därmed implementeras som var sin metod!

```
import javax.swing.*;
import java.util.*;
public class Cylinder {
  public static void main (String[] arg) {
    boolean done = false;
    while (!done) {
      String input = JOptionPane.showInputDialog("Ange radie och höjd:");
      if (input != null) {
        Scanner sc = new Scanner(input);
        double radius = sc.nextDouble();
        double height = sc.nextDouble();
        double area   = computeArea(radius, height);
        double volume = computeVolume(radius, height);
        JOptionPane.showMessageDialog(null, "Arean av cylindern är "+ area +
                                            "\nVolymen av cylindern är " +
                                            volume);
        } else {
          done = true;
       }
    }
  }
```

Lösning 2: fortsättning

```
    private static double computeArea(double radius, double height) {
        return 2 * Math.PI * radius * height +
               2 * Math.PI * Math.pow(radius, 2);
    }//computeArea

    private static double computeVolume(double radius, double height) {
        return Math.PI * Math.pow(radius, 2) * height;
    } //computeVolume

} //Cylinder2
```

Kommentar:

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

Lösning 3:

Arean av en cylinder beräknas med hjälp av cylinderns mantelyta samt cylinderns cirkelyta, och även volymen beräknas med hjälp av cirkelytan. Därför kan vi bryta ner problemen att beräkna cylinderns area och volym i ytterligare delproblem.

```
private static double computeArea(double radius, double height) {
    return computeSideArea(radius, height) +
           2 * computeCircleArea(radius);
}//computeArea

private static double computeVolume(double radius, double height) {
    return computeCircleArea(radius) * height;
} //computeVolume

private static double computeSideArea(double radius, double height) {
    return 2 * Math.PI * radius * height;
}//computeSideArea

private static double computeCircleArea(double radius) {
    return  Math.PI * Math.pow(radius, 2);
}//computeCircleArea
```

Lösning 4:

I föregående lösning har vi en klass som innehåller både ett huvudprogram och de *privata* klassmetoderna computeArea, computeVolume, computeSideArea och computeCircleArea. Det är även möjligt (och lämpligt) att lägga dessa metoder i en annan klass och än huvudprogrammet. Metoderna måste då göras *publika*.

```
public class Formulas {
     public static double computeArea(double radius, double height) {
          return computeSideArea(radius, height) +
                 2 * computeCircleArea(radius);
     }//computeArea

     public static double computeVolume(double radius, double height) {
          return computeCircleArea(radius) * height;
     } //computeVolume

     public static double computeSideArea(double radius, double height) {
          return  2 * Math.PI * radius * height;
     }//computeSideArea

     public static double computeCircleArea(double radius) {
          return  Math.PI * Math.pow(radius, 2);
     }//computeCircleArea
}//Formulas
```

Vad är fördelarna med denna design?

Lösning 4: fortsättning

```
import javax.swing.*;
import java.util.*;
public class Cylinder4 {
  public static void main (String[] arg) {
    boolean done = false;
    while (!done) {
      String input = JOptionPane.showInputDialog("Ange radie och höjd:");
      if (input != null) {
        Scanner sc = new Scanner(input);
        double radius = sc.nextDouble();
        double height = sc.nextDouble();
        double area   = Formulas.computeArea(radius, height);
        double volume = Formulas.computeVolume(radius, height);
        JOptionPane.showMessageDialog(null, "Arean av cylindern är " + area +
                                            "\nVolymen av cylindern är " +
                                            volume);
      } else {
        done = true;
      }
    }
  } //main
}//Cylinder4
```

Anm: För att programmet skall fungera måste klassen Formulas ligger i samma mapp som klassen Cylinder4.

OBS!

- När ni kopiera/klistra kod borde ett larm gå…
- Chans att abstrahera!!!

Paus

15 min

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

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

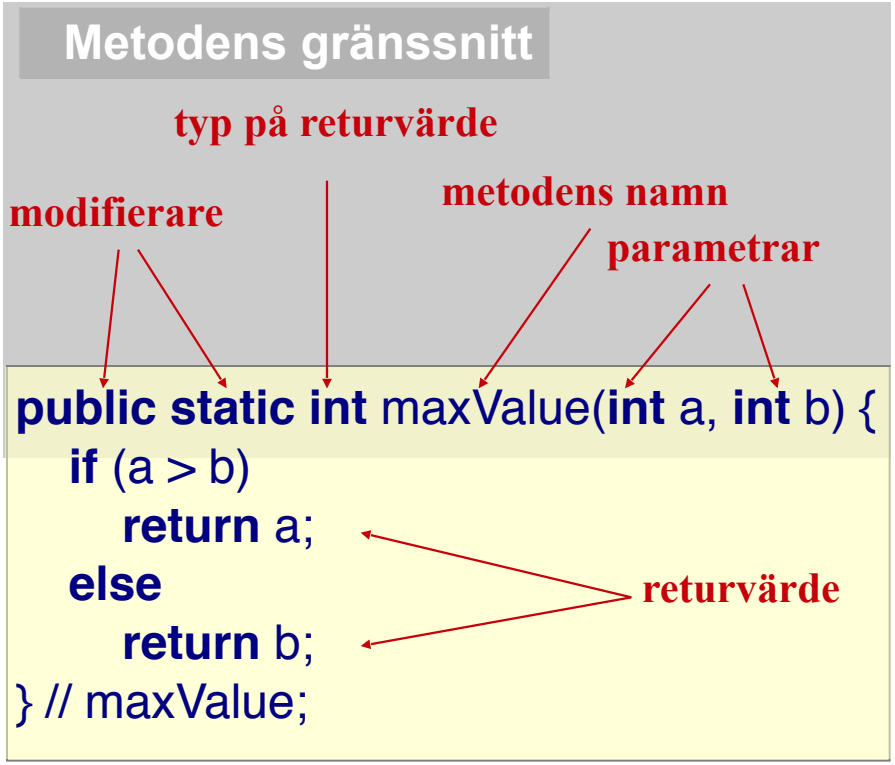
```
//Utseende på metoder som inte lämnar returvärde
 modifierare void namn(parameterlista) {
      dataattribut och satser
 }
```

Metoder kan antingen vara klassmetoder eller instansmetoder.

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

Metoder kan bl.a. vara **private** eller **public**.

Uppbyggnaden av en metod



Satsen

return *uttryck*;

terminerar metoden och värdet *uttryck* blir resultatet som erhålls från metoden.

En metod som inte lämnar något värde (en ***void**-metod*) har ingen **return**-sats eller har **return**-satser som saknar *uttryck*.

Uppbyggnaden av en metod

För att kunna använda en metod måste man känna till och kunna använda *metodens gränssnitt* på ett korrekt sätt. En metods gränssnitt bestäms av

- metodens *namn*
- metodens *returtyp*
- metodens *parameterlista* avseende antal parametrar samt parametrarnas typer och ordning
- huruvida metoden är en *klassmetod eller instansmetod*

Ett metodanrop kan ses som att en avsändare skickar ett meddelande till en mottagare. Parameterlistan beskriver vilken typ av data avsändaren kan skicka i meddelandet och resultattypen beskriver vilken typ av svar avsändaren får i respons från mottagaren.

Formella och aktuella parametrar

```
import javax.swing.*;
import java.util.*;
public class Exempel {
   public static int  maxValue(int a, int b) {
      if (a  > b)
         return a;
      else
         return b;
   } //maxValue

   public static void main(String[] args) {
      String input = JOptionPane.showInputDialog("Ge tre heltal");
      Scanner sc = new Scanner(input);
      int value1 = sc.nextInt();
      int value2 = sc.nextInt();
      int value3 = sc.nextInt();
      int big = maxValue(value1, value2);
      big = maxValue(big, value3);
      JOptionPane.showMessageDialog(null, "Det största av talen " + value1 + ", "
                                          + value2 + " och " + value3 + " är " + big);
   } // main
} //Exempel
```

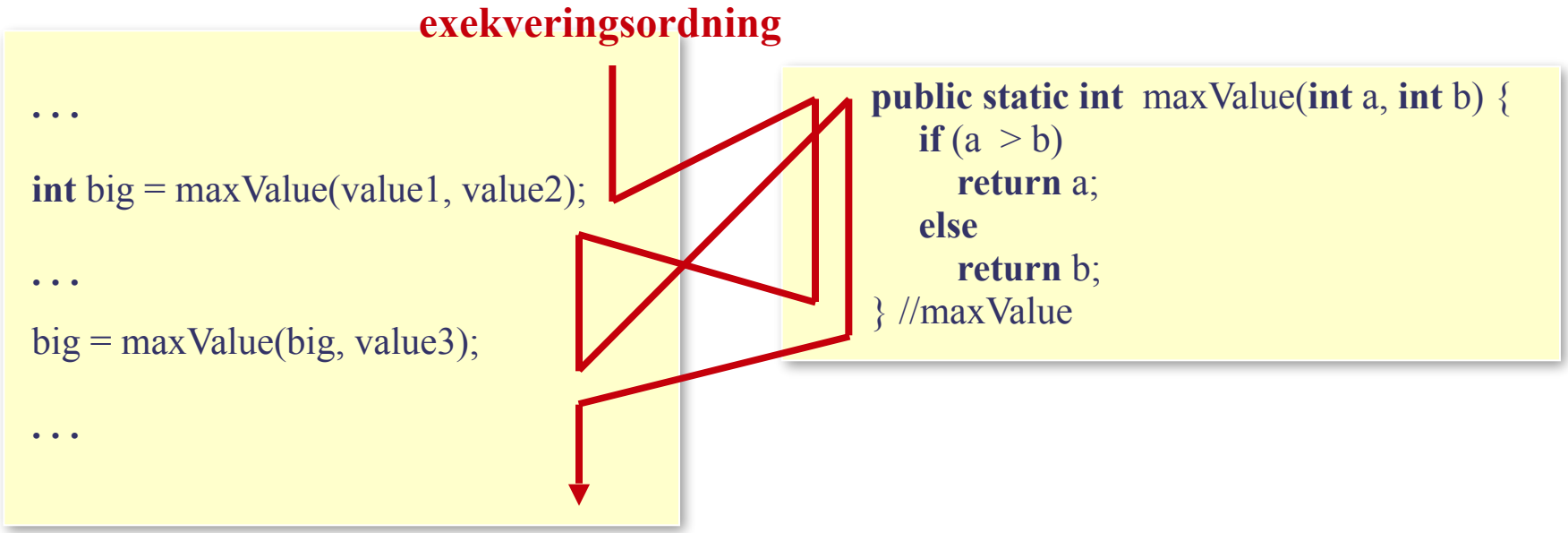
formella parametrar

aktuella parametrar

Metodanrop

Vid anrop av en metod sker följande:

- värdet av de aktuella parametrarna kopieras till motsvarande formell parameter
- exekveringen fortsätter med den första satsen i den anropade metoden
- när exekveringen av den anropade metoden är klar återupptas exekveringen i den metod där anropet gjordes



Parameteröverföring

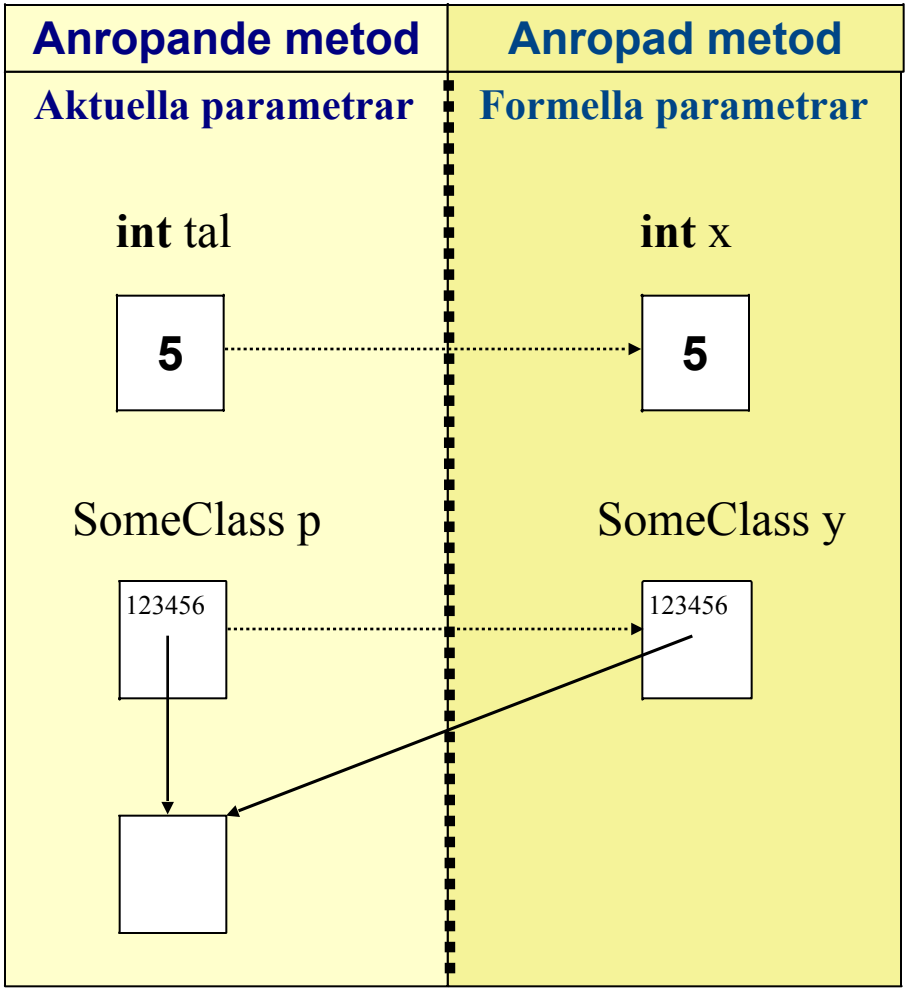
Alla primitiva datatyper och alla existerande klasser kan ges i parameterlistan och/eller som resultattyp.

Parameterlistan kan innehålla ett godtyckligt antal parametrar

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*.

En förändring av värdet i variabeln x påverkar inte värdet i variabeln tal.

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

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

En förändring *i objektet* som refereras av variabeln y påverkar därför objektet som refereras av variabeln p, eftersom det är samma objekt

Laboration 2

I laboration 2 skall ni programmera en robot som modelleras av den givna klassen Robot. En robot vistas i en enkel värld, och kan utföra några enkla operationer:

Följande operationer finns (samt några till):

void move() förflyttar sig ett steg framåt. Om roboten hamnar utanför världen eller i en mur fås ett exekveringsfel.

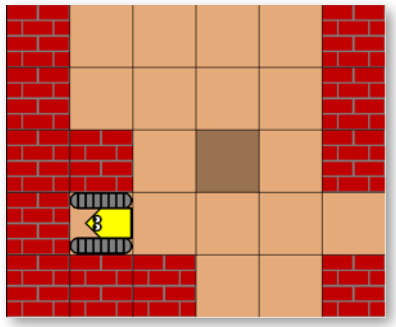
boolean frontIsClear() returnerar **true** om det är möjligt för roboten att göra move() utan att ett exekveringsfel erhålls, annars returnera **false**

void turnLeft() vrider sig 90° åt vänster

void makeLight() färgar rutan den står på till ljus. Om rutan redan är ljus fås ett exekveringsfel.

boolean onDark() returnerar **true** om roboten står på en mörk ruta, annars returneras **false**

int getDirection() returnerar robotens riktning.

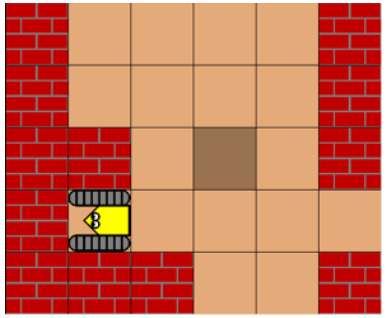


Syftet med laborationen är att ni bryta ner de uppgifter som roboten skall utföra i delproblem och utveckla kraftfullare abstraktioner än de operationer som roboten tillhandahåller. Dessa abstraktioner implementeras som små meningsfulla och återanvända metoder – med hjälpa av de operationer som roboten erbjuder.

Laboration 2

Roboten har ingen operation för att vrida sig runt, utan det måste görs med två på varandra följande anrop av den tillgängliga operationen turnLeft.

Antag att vi har en situation som i scenariot i bilden till vänster. Om vårt problem är att flytta roboten till den mörka rutan är turnAround en meningsfull abstraktion.

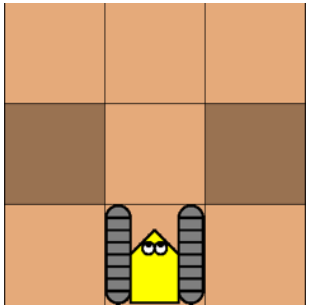


I laborationen är roboten en instansvariabel med namnet robot, varför abstraktionen turnAround skall implementeras som en instansmetod:

```
//before: none
//after: the robot is facing the opposite direction
public void turnAround() {
        robot.turnLeft();
        robot.turnLeft();
}//turnAround
```

Förvillkor och eftervillkor

Antag att vi har en situation som i scenariot i bilden till vänster. Vårt uppgift är att flytta roboten till den ljusa rutan som befinner ovanför roboten göra denna mörk. Detta kan vi göra med abstraktionen makeNorthLight.

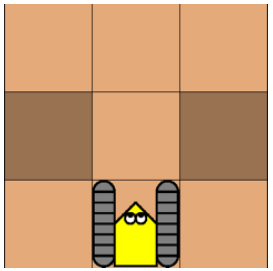


Implementationen får följande utseende:

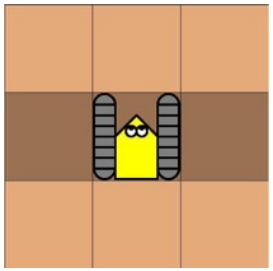
```
public void makeNorthDark() {
    robot.move();
    if (!robot.onDark())
        robot.makeDark();
}//makeNorthDark
```

Förvillkor och eftervillkor

Metoden (abstraktionen) fungera utmärkt för det scenario vi utgick ifrån:

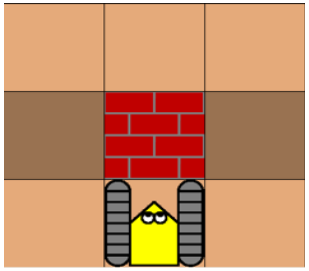


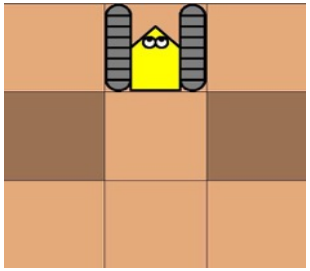
Före:

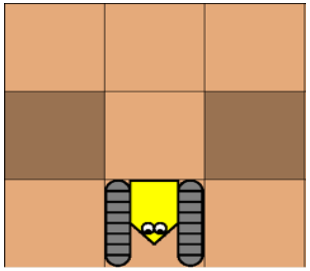


Efter:

Men det finns flera scenarior där metoden inte fungerar, varav några framgår av bilderna nedan:







Förvillkor och eftervillkor

När vi använder en metod måste vi veta vilka antaganden som görs i implementationen av metoden, d.v.s. vad som måste gälla för att metoden skall fungera på avsett sätt. Dessa antaganden kallas för *förvillkor* och måste specificeras.

Det är också viktigt att veta om metoden har några biverkningar (*sidoeffekter*) då den utförs. Syften med metoden makeNorthDark är att flytta roboten till rutan ovanför och göra denna ruta mörk. Men vilken riktning har roboten när metoden utförts? Detta är en sidoeffekt som behöver specificeras i ett *eftervillkor*.

```
//before: the robot is facing north, with no wall in
//           front or not at the edge of the world
//after: the cell to north is dark, the robot is in
           this cell facing north
public void makeNorthDark() {
    robot.move();
    if (!robot.onDark())
       robot.makeDark();
 }//makeNorthDark
```

Förvillkoren till en metod måste vara kontrollerbara för den som anropar metoden. I detta fall kan förvillkoret kontrolleras med metoden frontIsClear().

Förvillkor och eftervillkor

```
public class Formulas {
    //before: radius >= 0 && height >= 0
    //return: the area of a cylinder with assigned raduis and height
    public static double computeArea(double radius, double height) {
       return  computeSideArea(radius, height) + 2*computeCircleArea(radius);
    }//computeArea

    //before: radius >= 0 && height >= 0
    //return: the volume of a cylinder with assigned raduis and height
    public static double computeVolume(double radius, double height) {
       return computeCircleArea(radius) * height;
    } //computeVolume

    //before: radius >= 0 && height >= 0
    //return: the side area of a cylinder with assigned raduis and height
    public static double computeSideArea(double radius, double height) {
       return  2*Math.PI*radius*height;
    }//computeSideArea

    //before: radius >= 0
    //return: the area of a circle with assigned raduis
    public static double computeCircleArea(double radius) {
       return  Math.PI*Math.pow(radius, 2);
    }//computeCircleArea
}//Formulas
```

En cylinder kan inte ha en radie som är negativ och inte en höjd som är negativ.

En cirkel kan inte ha en radie som är negativ.

Förvillkor och eftervillkor

För att en programmerare skall kunna använda sig av en metod på ett korrekt sätt måste han/hon känna till *specifikationen* för en metoden:

- metodens namn
- metodens parameterlista
- metodens returtyp
- vad metoden gör
- vilka förvillkor som måste gälla
- vilka eftervillkor (sidoeffekter) metoden har.

För att kunna använda en metod behöver man däremot *inte* känna till hur metoden är implementerad. Det intressanta är *vad* metoden gör *inte hur* den gör det!

Specifikationen är alltså viktig att dokumentera!!

javadoc

```
public class Formulas {
    /**
          @before radius >= 0 && height >= 0
          @return the area of a cylinder with assigned raduis and height
    */
     public static double computeArea(double radius, double height) {
        return  computeSideArea(radius, height) + 2*computeCircleArea(radius);
     }//computeArea
     /**
          @before radius >= 0 && height >= 0
          @return the volume of a cylinder with assigned raduis and height
    */
     public static double computeVolume(double radius, double height) {
        return computeCircleArea(radius) * height;
     } //computeVolume
      /**
          @before radius >= 0 && height >= 0
          @return the side area of a cylinder with assigned raduis and height
     */
     public static double computeSideArea(double radius, double height) {
        return  2*Math.PI*radius*height;
     }//computeSideArea
     /**
          @before radius >= 0
          @return the area of a circle with assigned raduis
    */
     public static double computeCircleArea(double radius) {
        return  Math.PI*Math.pow(radius, 2);
     }//computeCircleArea
}//Formulas
```

@return är en fördefinierad annotation

@before är en *egendefinierad* annotation

Kommandot

javadoc -tag before:a:"Before:" Formulas.java

skapar en dokumentation av klassen Formulas i form av en uppsättning html-filer.

---

computeArea

```
public static double computeArea(double radius,
                                 double height)
```

Returns:
the area of a cylinder with assigned raduis and height
Before:
radius >= 0 && height >= 0

---

computeVolume

```
public static double computeVolume(double radius,
                                   double height)
```

Returns:
the volume of a cylinder with assigned raduis and height
Before:
radius >= 0 && height >= 0

Exempel på andra fördefinerade annotationer:

@author
@before
@version
@exception
@param