Objektorienterad programmering

Föreläsning 4: metoder och top-down design

Dr. Alex Gerdes | Dr. Carlo A. Furia

Hösttermin 2016

Chalmers University of Technology

Några meddelande

- Bara några dagar kvar till deadlinen för laboration 1
- Läs textboken, läsanvisningar finns på hemsidan
- Kom ihåg *flödet*: läsa, föreläsning, öva/lab
- Finns extra instuderingsfrågor på hemsidan
- Jätte bra med frågorna, keep-em-coming!

Sammanfattning föreläsning 3

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

Abstraktion

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.

THE LIFE OF A SOFTWARE ENGINEER.

CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.

MUCH LATER...

OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

MANU

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

Top-down design

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

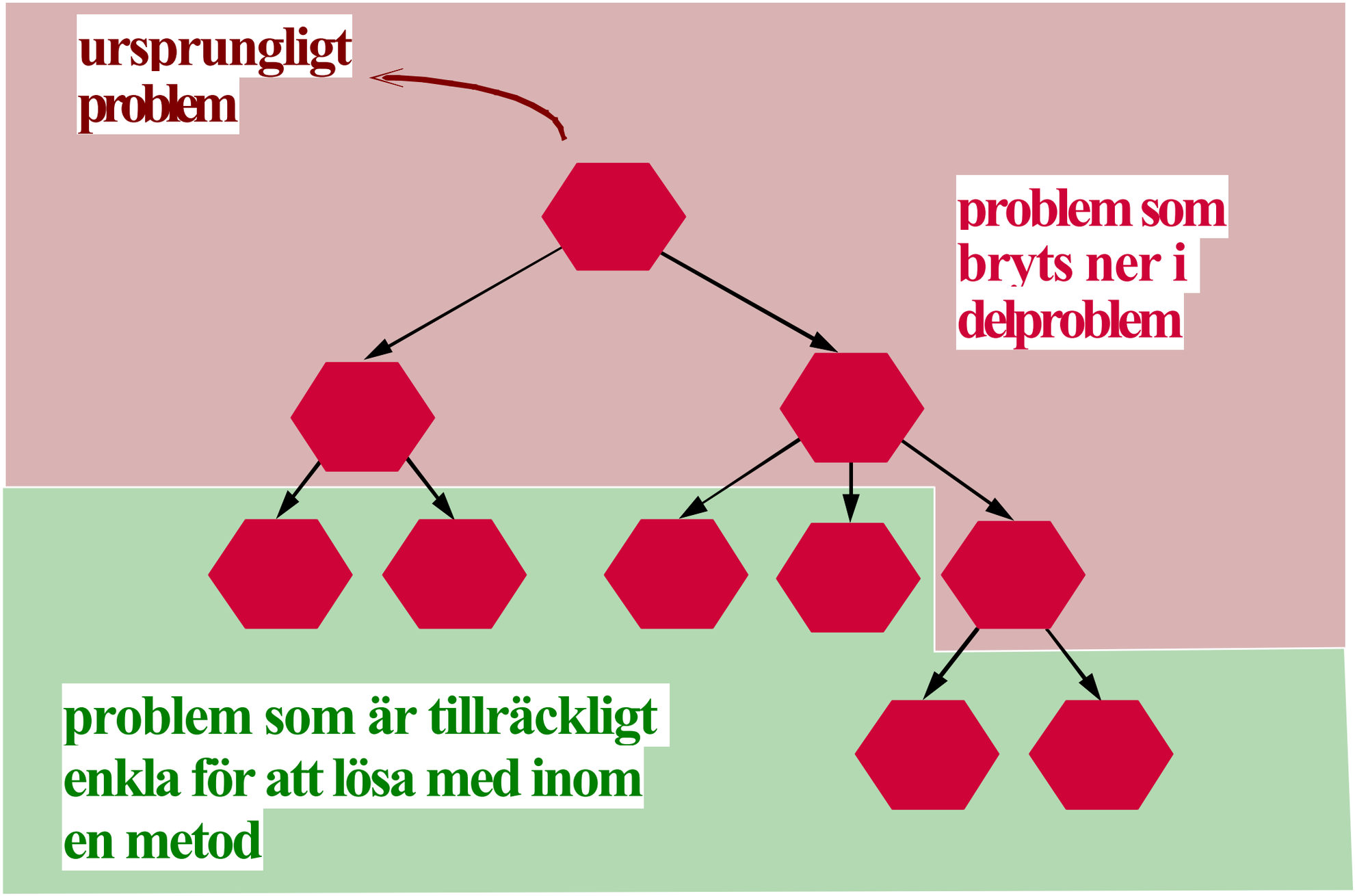
- Top-down design innebär att vi betraktar det ursprungliga problemet på en *hög abstraktionsnivå* och bryter ner det ursprungliga problemet i ett antal delproblem

- 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*

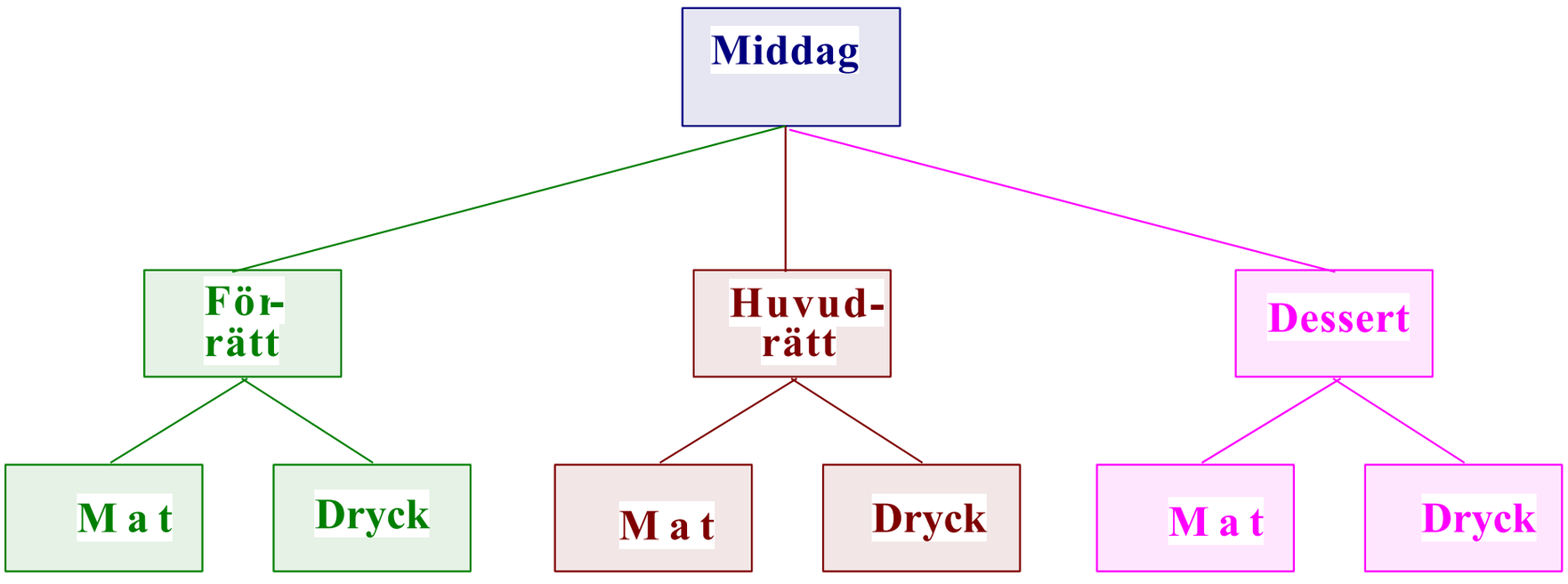
Top-down design



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"



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

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

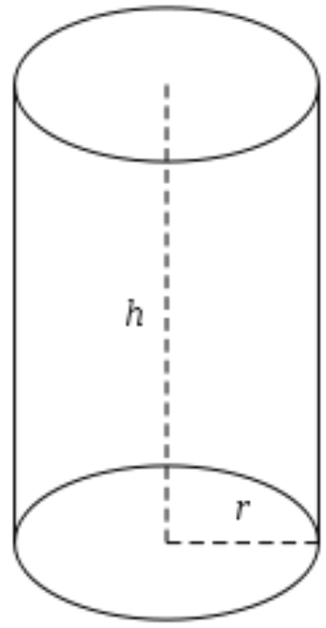
Designa programsystemet runt *stabila abstraktioner* och *utbytbara* komponenter för att möjliggöra små och stegvisa förändringar.

Exempel

- **Problem**: 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:

$$A = 2\pi r h + 2\pi r^2 \qquad V = \pi r^2 h$$

där r är radien och h är höjden av cylindern



- **Algoritm**:
  1. Läs cylinderns radie r
  2. Läs cylinderns höjd h
  3. Beräkna cylinderns area A mha formeln $A = 2\pi r h + 2\pi r^2$
  4. Beräkna cylinderns volym V mha formeln $V = \pi r^2 h$
  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 CalcCylinder {
  public static void main(String[] args) {
    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);
    }
  }
}
```

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 CalcCylinderV2 {
  public static void main(String[] args) {
    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);

    }
  }
```

Anropa metoder för att utföra beräkningarna

Lösning 2, fortsättning

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

    private static double computeVolume(double radius, double height) {
        return Math.PI * Math.pow(radius, 2) * height;
    }
} // Slut av klassen CalcCylinderV2
```

- **Kommentar**: vi har deklarerat metoderna `computeArea` och `computeVolume` som `private` för 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);
}

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

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

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

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, externa metoderna måste då göras *publika*

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

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

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

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

Vad är fördelarna med denna design?

Lösning 4, fortsättning

```
import javax.swing.*;
import java.util.*;

public class CalcCylinderV4 {
  public static void main(String[] args) {
    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   = Cylinder.computeArea(radius, height);
      double volume = Cylinder.computeVolume(radius, height);
      JOptionPane.showMessageDialog(null, "Arean av cylindern är " + area +
                                          "\nVolymen av cylindern är " +
                                          volume);
    }
  }
}
```

Anm: för att programmet skall fungera måste klassen `Cylinder` ligger i samma mapp som klassen `CalcCylinderV4`

Observera!

- När ni kopiera/klistra kod borde ett larm gå...

Chans att abstrahera!

Paus (15 min)

Metodanrop

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`

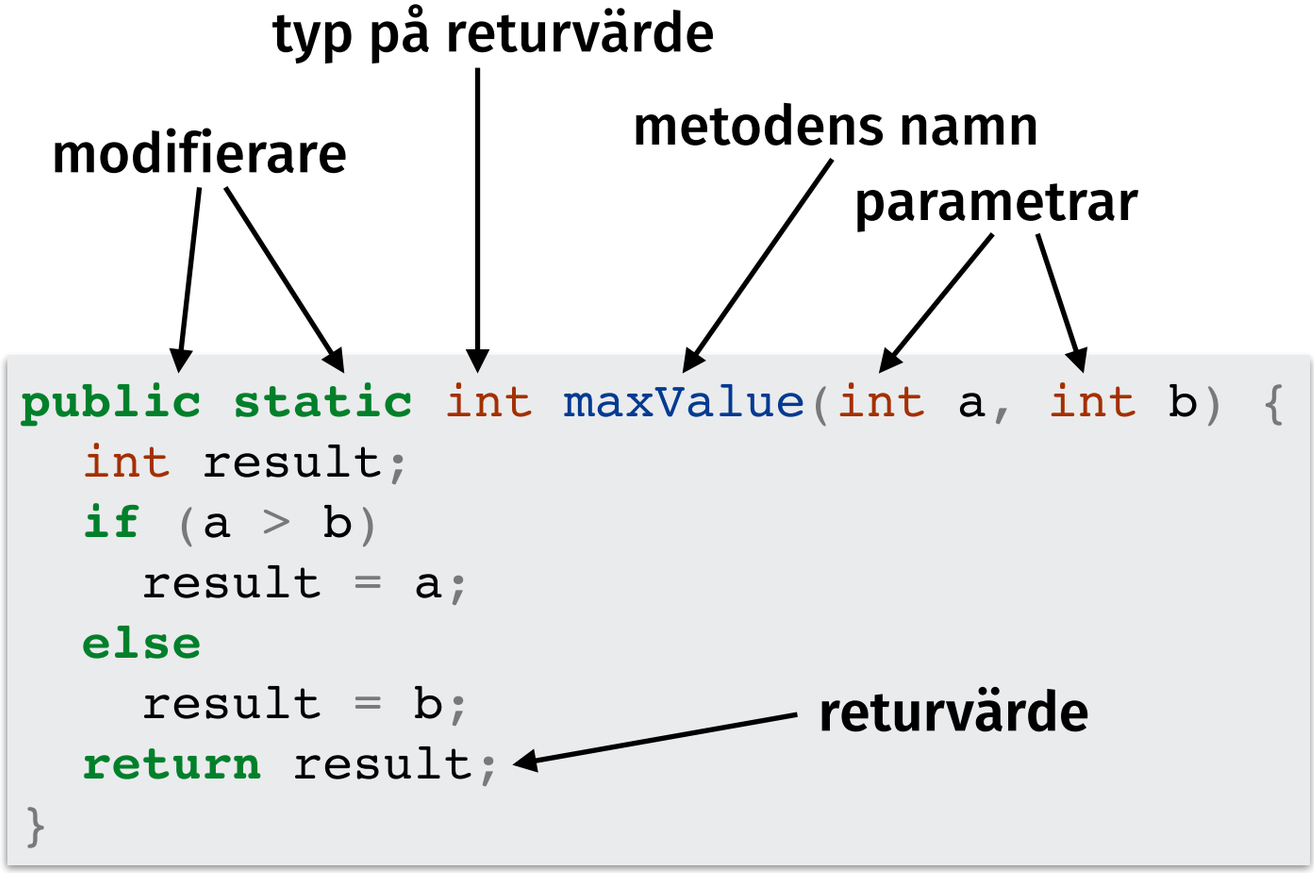
Metodens gränssnitt

- 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

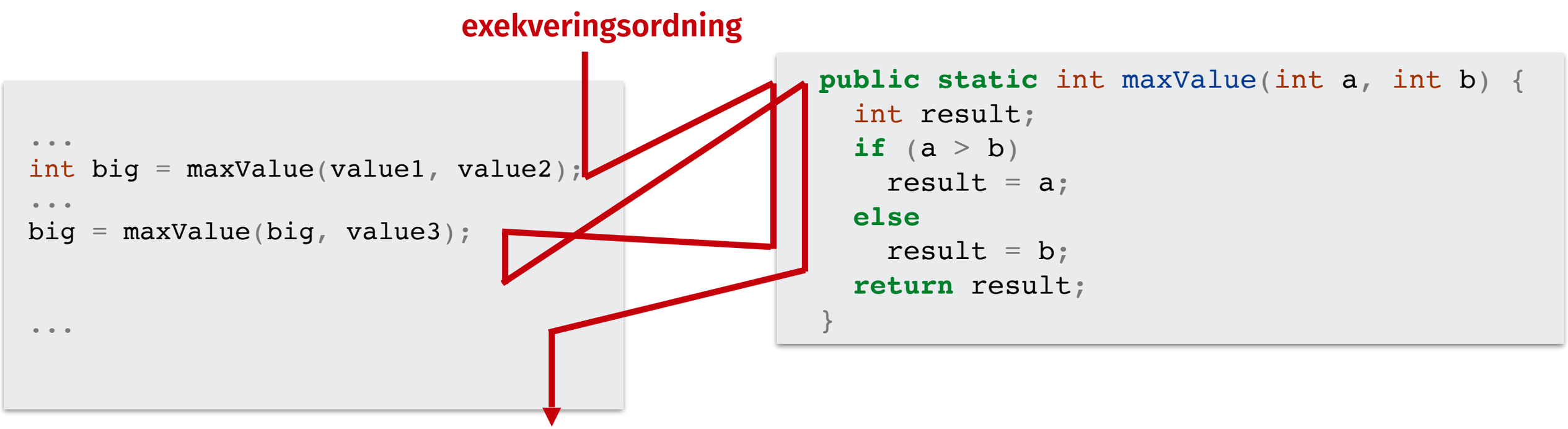
```
import javax.swing.*;
import java.util.*;

public class Exempel {
  public static 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);
  }

  public static int maxValue(int a, int b) {
    int result;
    if (a  > b)
      result = a;
    else
      result = b;
    return result;
  }
}
```

aktuella parametrar

formella 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

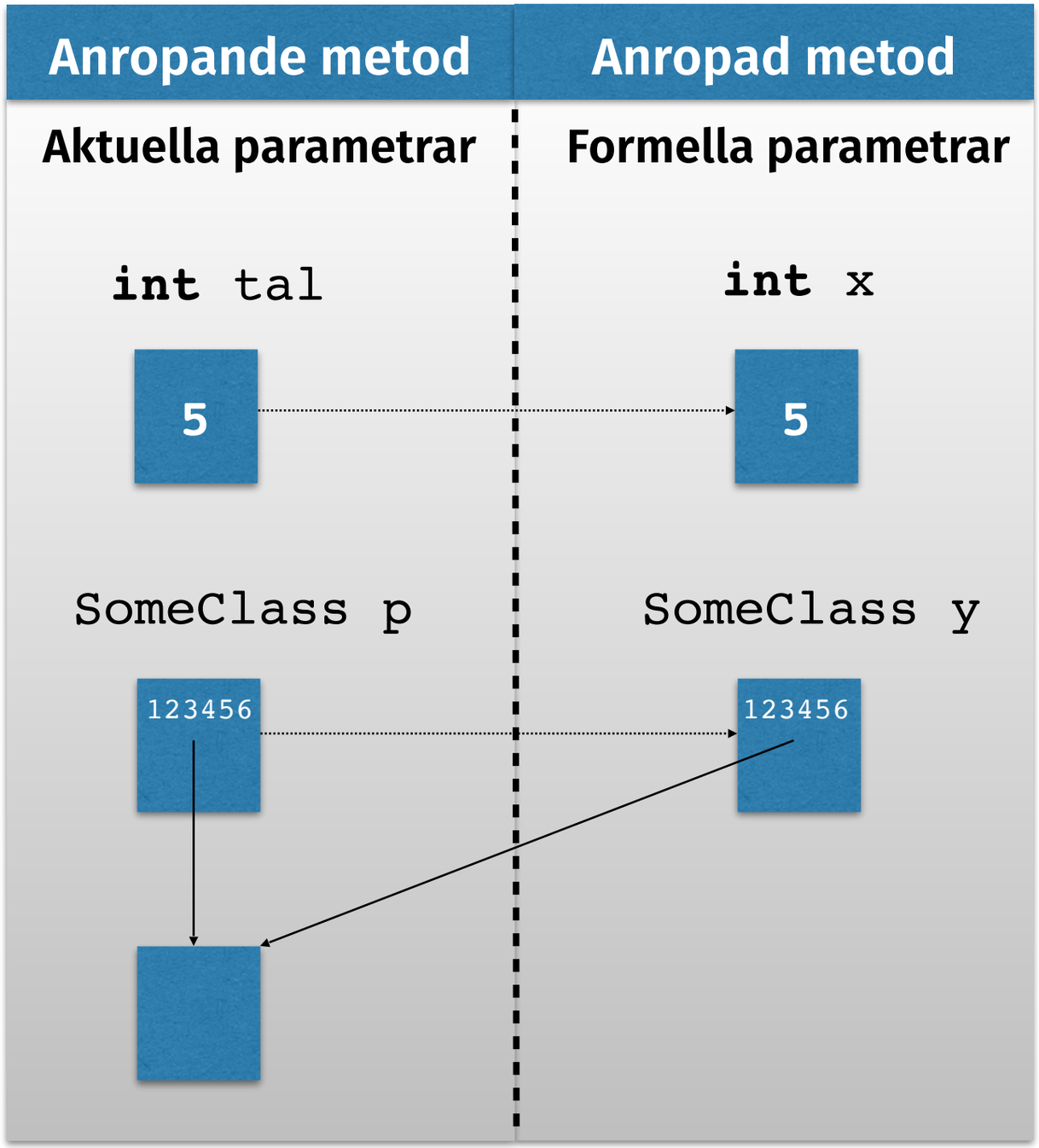


Parameteröverföring

- 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`

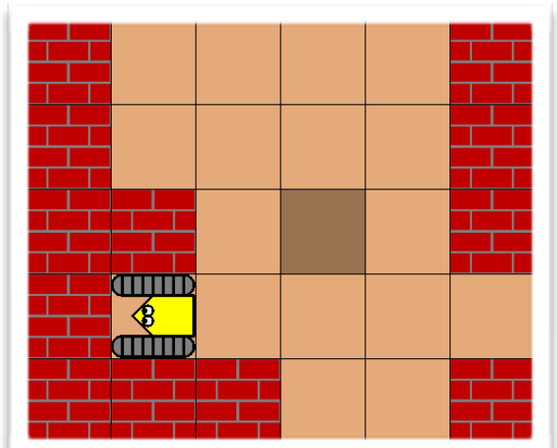


- Värdet av den aktuella parametern `p` kopieras till den formella parametern `y`
- `p` och `y` kommer att referera till *samma* fysiska objekt
- En förändring i objektet som refereras av variabeln `y` påverkar därför objektet som refereras av variabeln `p`, eftersom det är samma objekt

Laboration 2

Laboration 2

- I laboration 2 skall ni programmera en robot som modelleras av den givna klassen `Robot`

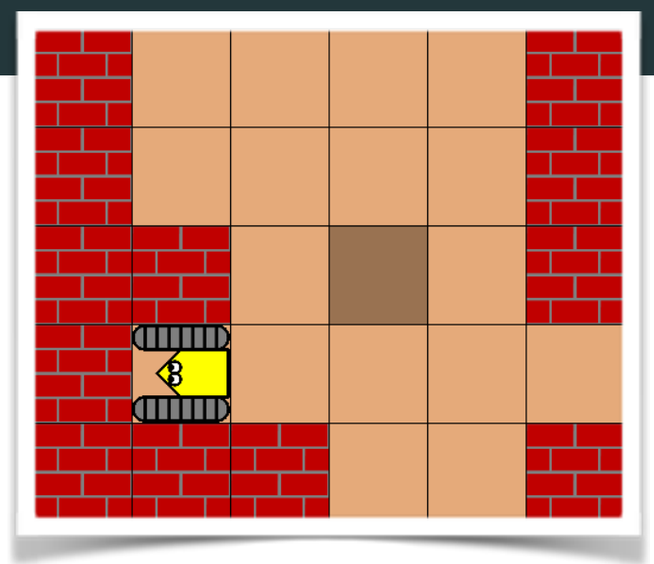


- Syftet med laborationen är att ni bryta ner de uppgifter som roboten skall utföra i delproblem och utveckla *kraftfullare abstraktioner* än de operationer som roboten tillhandahåller; dessa abstraktioner implementeras som små meningsfulla och återanvända metoder – med hjälpa av de operationer som roboten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

- En robot vistas i en enkel värld, och kan utföra några enkla operationer (samt några till):

Metod	Användning
`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

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`

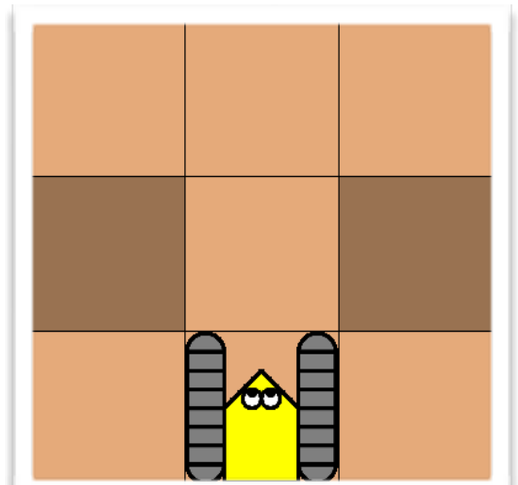
- Antag att vi har en situation som i scenariot i bilden; mm vårt problem är att flytta roboten till den mörka rutan är `turnAround` en meningsfull abstraktion

- 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();
}
```

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 framför roboten göra denna mörk. Detta kan vi göra med abstraktionen `moveAndDarken`.

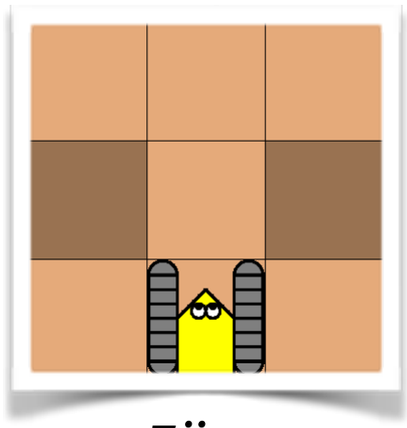


- Implementationen får följande utseende:

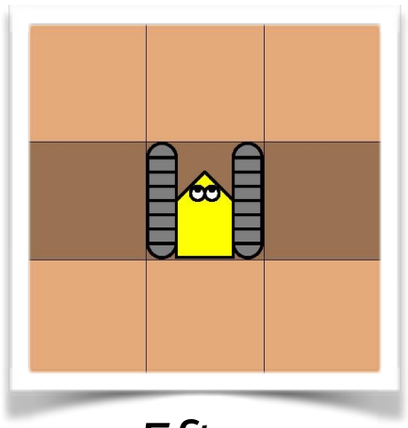
```
public void moveAndDarken() {
  robot.move();
  if (!robot.onDark())
    robot.makeDark();
}
```

Förvillkor och eftervillkor

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

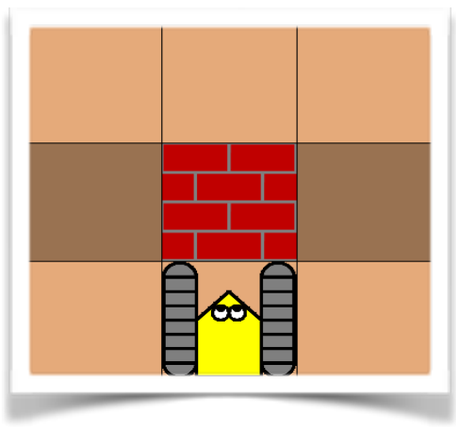


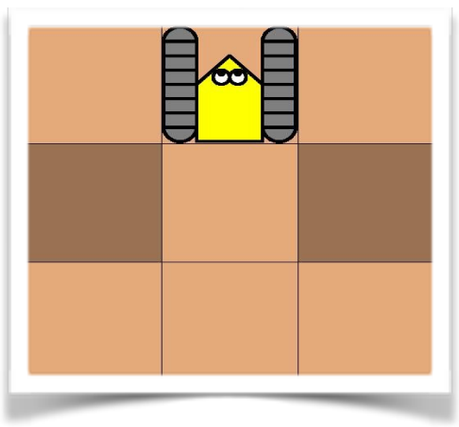
Före

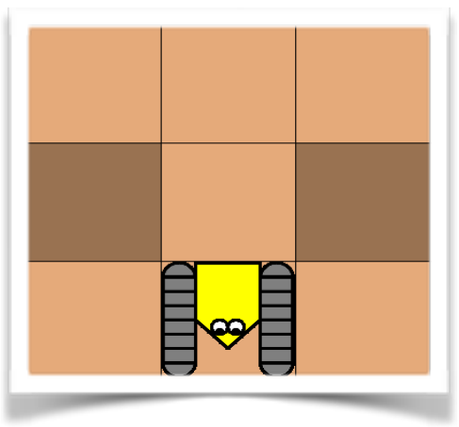


Efter

- 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 `moveAndDarken` 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*.

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

```
// before: the robot is not facing a wall or at the
//         edge of the world
// after:  the cell in front is dark, and the robot moved to that
//         cell and has not changed direction
public void makeNorthDark() {
  robot.move();
  if (!robot.onDark())
      robot.makeDark();
}
```

Förvillkor och eftervillkor

- 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

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

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

- `@return` är en fördefinierad annotation
- `@before` är en egendefinierad annotation
- Kommandot

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

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

- Exempel på andra fördefinerade annotationer:
  - `@author`
  - `@before`
  - `@version`
  - `@exception`
  - `@param`

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

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

Live coding: laboration 2