Programkonstruktion

TDA548/Joachim von Hacht

Happy Hacking

```
384
385
386             if(!scene6TF1.getText().isEmpty() && scene6TF1.getText().length() < 21 && Pattern.matches("[\306\330\305\346\370\34
387                 if (!scene6TF2.getText().isEmpty() && scene6TF2.getText().length() < 51 && Pattern.matches("[\306\330\305\346\3
388                     if (!scene6TF3.getText().isEmpty() && scene6TF3.getText().length() < 81 && Pattern.matches("[\306\330\305\3
389                         if (!scene6TF4.getText().isEmpty() && scene6TF4.getText().length() == 8 && Pattern.matches("[0-9]+", sc
390                             if (!scene6TF5.getText().isEmpty() && scene6TF5.getText().length() == 4 && Pattern.matches("[0-9]+",
391                                 if (!scene6TF6.getText().isEmpty() && scene6TF6.getText().length() == 10 && Pattern.matches("[0
392                                     if(scene6CB1.getValue() != null) {
393                                         if(scene6CB2.getValue() != null)
394                                         {
395                                             System.out.println("Tillykke, alle felter er godkendt!");
396                                             // Opret medarbejder - det vil sige foretag et SQL statement der indsætter de forskı
397
398                                         } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Løntrin er ikke valgt i drop-down menu
399                                     } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Stilling er ikke valgt i drop-down menuen."
400                                 } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Kontonummeret indtastet forkert. Feltet må kun
401                             } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Registreringsnummer indtastet forkert. Feltet må ku
402                         } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Telefonnumer indtastet forkert. Feltet må kun indeholde
403                     } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Adresse indtastet forkert. Feltet må kun indeholde bogstave
404                 } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Efternavn indtastet forkert. Feltet må kun indeholde bogstaver,
405             }  else { FejlMeddelelse.visFejlMeddelelse("Fejl: Fornavn indtastet forkert. Feltet må kun indeholde bogstaver, mel
406         });
```


För att skapa begripliga program måste man konstruera dem på ett bra sätt

- Det räcker inte att “koda på” (happy hacking) tills det eventuellt (troligen inte) fungerar.

Happy hacking leder till program som

- Inte går att felsöka
- Inte går att förändra
- Inte går att bygga på

Vi går igenom ett antal grundläggande sätt att arbeta och konstruera program. Mycket mer i senare kurser.

- Se vidare programvaruutveckling (software engineering)

(Av)kommentera och TODO

```
/*coin = new ImageIcon(ImageIO
        .read(new
File("src/exercises/optional/gold_coin_single.png")))
        .getImage();*/
coin = new ImageIcon(this.getClass() // TODO better use Image?

        .getResource("gold_coin_single.png"))
        .getImage();
// getAudioInputStream() also accepts a File or InputStream
//AudioInputStream ais = AudioSystem.
// getAudioInputStream(new
File("src/exercises/optional/atari.wav"));
AudioInputStream ais = AudioSystem
     .getAudioInputStream(this.getClass()
     .getResourceAsStream("atari.wav"));
```

3

Använd kommentarer istället för att ta bort kod!

- Lätt att få tillbaka den gamla koden om den visade sig vara bra!

Ofta stöter man på saker som man borde fixa till (när man egentligen håller på med en annan sak).

- Ta för vana att lägga in TODO noteringar (sökbara i IntelliJ)

I bilden

- Gammal kod bortkommenterad (tills vi är säkra på att den nya är bättre, annars, kommentera ut den nya och avkommentera den gamla)
- TODO-notering för att senare kolla om Image är bättre än ImageIcon

Dessutom: Ctrl z i IntelliJ (undo)

Minsta Steget

```
void program() {
   // Very small basic step
   out.println("Program started");

}
```

4

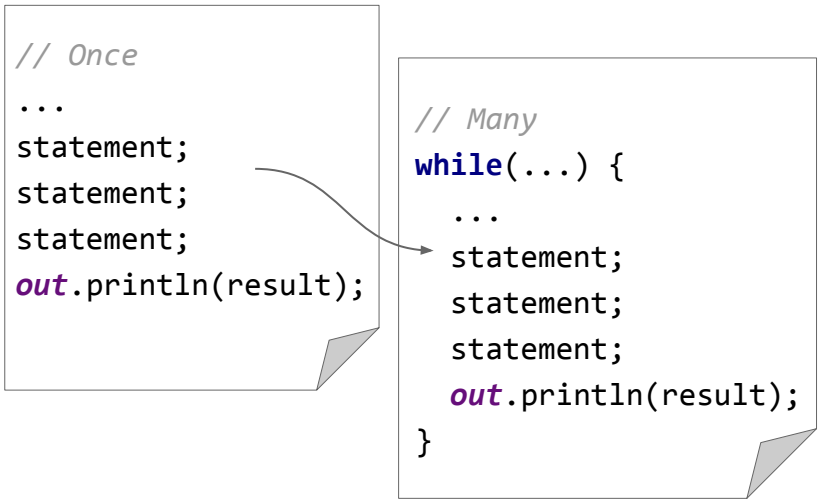
Ett arbetssätt!

När vi skriver ett program (eller håller på med en del av) kan man inte skriva allt på en gång!

- Man börjar med att försöka hitta minsta möjliga steg som för oss närmare lösningen
- Kodar detta och kontrollerar om det fungerar.
- Därefter nästa steg o.s.v.
- Detta gäller generellt (inte bara i denna kurs)!

OBS! Skall alltid ha något körbart!

Flera Gånger

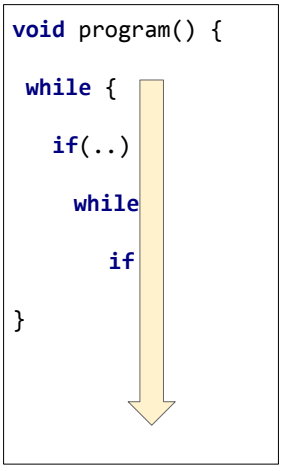


5

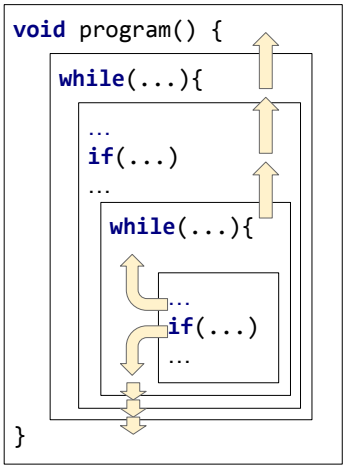
Ännu ett arbetssätt.

- Om man kan göra något en gång, … är det lätt att göra det flera gånger … lägg en loop runt!
- Börja med att göra något en gång först!

Inifrån Ut



Uppifrån och ner



Inifrån ut

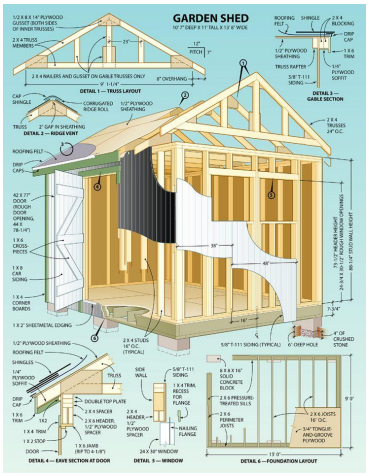
6

"Inifrån ut" innebär att man inte skriver programmet "rad för rad" uppifrån och ned.

Istället börjar man "inifrån" med "minsta steget".

- Några satser som gör det mest elementära.
- När man fått till detta utökar man programmet genom att lägga till ev. if/while etc. som ligger "runt" det elementära
- När detta fungerar utökar man med nästa "lager" o.s.v.

Ritning, Plan och Arbetssätt



7

En analogi inför laborationerna (som är lite större program).

Om vi skall bygga ett uthus måste vi ha en ritning
- Vi kan inte börja såga och spika innan vi vet vad som skall sågas och spikas!

När vi har en ritning måste vi bestämma i vilken ordning vi skall göra saker, vi gör en plan.
- Vi gör klar de olika momenten i en viss ordning
  - För varje moment tillämpar vi ett arbetssätt
    - Vi gör klart ett moment m.h.a av de verktyg vi har och den hantverksskicklighet vi besitter (tumregler, snickarknep)
    - Vi kontrollera allt verkar stämma (t.ex. med vattenpass, vinkelhake och lod)
- Därefter nästa moment

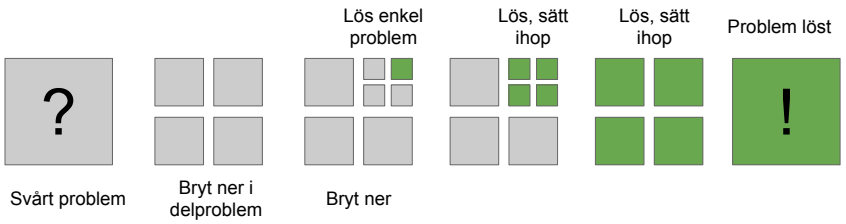
Om vi skall skriva ett lite större program måste vi också ha en “ritning”
- Vi kan inte börja koda förrän vi har en aning om vad vi vill åstadkomma
- Att skapa en ritning för ett program betyder att på papper skriva/rita/kladda ner “något”.

        - Vi gör detta mycket informellt, mer strax ...

Därefter bestämmer vi i vilken ordning programmet skall implementeras (åtminstone var vi skall börja), vi gör en (skissartad) plan?

- Vi gör ett moment i taget utifrån i planen
    - För varje moment använder vi ett visst arbetssätt (t.ex. minsta steget)
        - Vi gör klart momentet med de konstruktioner vi känner till och med hjälp av den programmeringsskicklighet vi besitter.
        - Efter varje moment testar vi om det fungerar
- Därefter nästa moment.

Bryta ner Problem



8

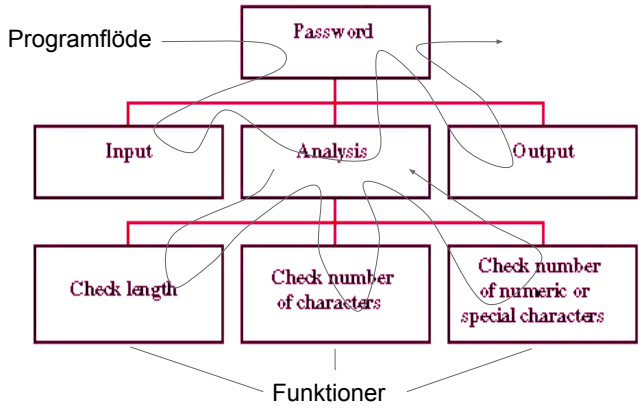
Att skriva program innebär att lösa (stora) problem.

Ett klassiskt angreppssätt om man har ett stort/svårt problem.

- Bryt ner i mindre/enklare problem!
- Lös de enklare problemen (eller dela dessa ytterligare)
- Sätt ihop alla lösningar av delproblemen till en lösning för hela problemet.

Konkret gör vi detta genom att använda funktionell nedbrytning.

Funktionell Nedbrytning



10

Funktionell nedbrytning (functional decomposition) innebär att man:

- Antar att man har en metod som löser hela problemet (i bilden: kontrollera lösenord, Password)
- Börjar skriva denna … när man stöter på ett nytt problem antar man att man har en metod som löser detta (Input, Analysis och Output).
- Därefter skriver man de metoder man antog man hade, behövs fler metoder antar man att man har dem (Check length, … o.s.v… tills metoderna man behöver är triviala (då implementerar man dem).
    - Genom att kombinera små enkla metoder har man löst ett stort problem. Klart!
- Funktionell nedbrytning är en top-down strategi. Man börjar med helheten och bryter ner i enklare delar.

Under arbetet med funktionell nedbrytning använder man **funktionell abstraktion**, en tankeform där man bara fokuserar på:

- Indata (vad har jag?)
- Utdata (vad vill jag ha?)
- … detta för att slippa alla detaljer om hur det skall gå till …
- Sikta på vad som skall göras inte hur!
    - Vi måste veta vad som skall göras innan vi börjar fundera på hur det skall göras.
- Abstraktion är ett ord ni kommer att stöta på mycket, ett försök att

- förklara abstraktion finns här

Genom att använda funktionell nedbrytning (plus anteckningar/skisser) får vi dessutom en ritning till programmet.

Metodstubbar

```
// Method under development
Player getPlayerLeft(Player[] players, Player actual) {
   int i = indexOf(players, actual);
   return players[(i + players.length - 1) % players.length];
}

// Stub with hard coded return (and todo note)
int indexOf(Player[] players, Player player) {
    return 0;   // TODO
}
```

10

Då man arbetar med funktionell nedbrytning händer det t.ex. att man upptäcker flera metoder samtidigt.

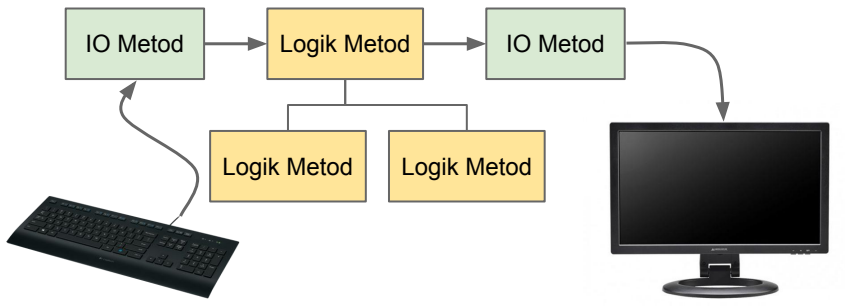
- Eftersom vi bara kan implementera en metod i taget så skriver man "stubbar" för de övriga

En **metodstubbe** är en "tom" metod.

- Genom att använda stubbar kan vi gör anrop till metoder som inte är färdiga d.v.s programmet kompilerar och går att köra (men resultatet blir inte rätt)
- Genom att använda stubbar får vi upp (kan köra) "strukturen" på lösningen.
- Om metoden returnerar ett värde får vi skriva dit något tillfälligt returvärde
    - Vanligen: return 0, return true eller return null

Detta är en viktig del i vårt arbetssätt!

Logik- och IO-Metoder



11

Vi skiljer på IO (programmet "utseende") och logik

- En logik-metod använder aldrig IO
  - Aldrig strömmarna in och out eller grafik.
  - Tilldelningar, beräkningar, logiskt flöde sköts här (det som inte syns)
- En IO-metod utför ingen programlogik
  - IO metoder läser in eller skriver ut data, visar bilder.
    - Finns det if och while här är dessa bara till för att utseendet skall bli på ett visst sätt, ingen annat
  - Metoden använder t.ex. strömmar (eller annat) för att läsa in eller mata ut.
  - IO metoder för utmatning är normalt void.

Testning

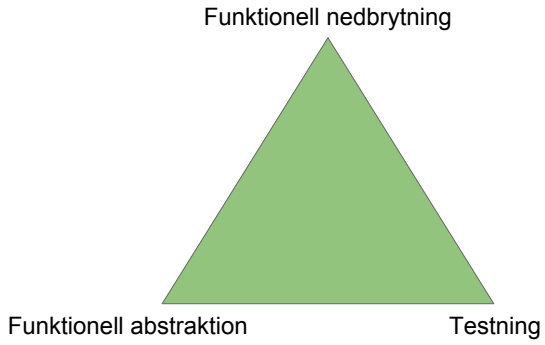
```
int d = rollDice();
out.println(1 <= d && d <= 6);   // true?

String[] r = roll(2);  // Only 2 dices so one of them should be empty
out.println(r[0].equals(EMPTY) ||
         r[1].equals(EMPTY) || r[2].equals(EMPTY)); // true?

Player[] players = {new Player("0lle", 3), new Player ...}; // 3 players
out.println(getPlayerLeft(players, players[0]) == players[2]);  // true?

String[] res = {"L", "C", "R"};
actual = players[1];
distributeChips(res, players, actual); // Initially all have 3 chips
out.println(players[0].chips == 4 &&
            players[1].chips == 0 && players[2].chips == 4); // true?
```


Vår teststrategi är följande:

- Så fort vi implementerat en metod så testar vi den.
- IO-metoder testar vi manuellt, vi provkör helt enkelt
  - IO metoder är (skall vara) enkla.
- För logikmetoder automatiserar vi testerna genom att skriva ut (out.println) ett uttryck som skall ge värdet true
  - I uttrycket gör vi en jämförelse mellan resultatet av ett metodanrop och ett förväntat korrekt värde (som vi måste veta)
    - Här krävs viss kreativitet för att komma på vilka jämförelser vi vill göra
    - Testerna måste vara så enkla som möjligt, vi vill inte introducera nya fel i själva testerna.
  - Vi skriver alla tester i en egen metod med namnet test() eller …
    - … senare i en egen klass men namnet Test.
  - Vi behåller alltid alla tester, när som helst kan vi köra dessa igen!

I senare kurser får du lära dig att använda speciella ""testramverk".

Sammanfattning (så långt)



13

En ritning, en plan och ett visst arbetssätt skall hjälpa er att klara kursens laborationer

- Genom att arbeta på ett visst sätt ökar vi vår förmåga att lösa problem (skriva program).

Funktionell nedbrytning: Förenklar problemet och ger en ritning till programmet

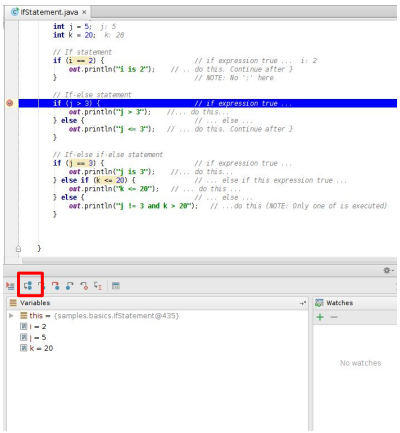
- Under nedbrytningen används funktionell abstraktion

Testning är ett mycket viktigt inslag i vårt arbetssätt

- Kan inte få ihop ett program om delarna inte fungerar.

Felsökning

Avlusare (debugger)



Utskrift av värde

```
out.print(someValue);
```

16

En stor del av programmeringsarbetet består av felsökning. Det finns flera alternativ

- Använda out.println() och skriv ut värden
  - Kan var en bra metod för upprepad inspektion
    - Utskriften sker varje gång programmet körs
  - Nackdelar:
    - Blir det för många utskrifter kan man till slut inte hänga med
    - Utskrifterna måste tas bort
- Använd en avlusare (Debugger)
  - En avlusare är ett program som kan köra ett annat (ditt) program sats för sats
  - Finns inbyggd i IntelliJ

Avlusning (procedur)

- Klicka i vänstermarginalen för att få en brytpunkt (röd prick ovan).
  - Klicka igen om du vill ta bort..
- Högerklicka i kodfönstret och välj Debug …
- Avlusaren startar och kör programmet fram till brytpunkten. Där stannar det.
- Därefter kan du köra sats för sats genom att klicka Step Over (röd fyrkant i bilden)

- Den blå raden skall hoppa en sats då du klickar
- Du kan hela tiden inspektera variabelvärden i det nedre fönstret
- Avsluta genom att klicka röd fyrkant ner till vänster (visas ej)
- Hakar något upp sig … börja om

Redundant Kod

```
// Bad redundant code!
if ( … ) {
    …
    dices = 1;
} else {
    …
    dices = 1;
}
```

```
// Non redundant
if ( …  ) {
   …
} else {
    …
}
dices = 1;
```

15

Redundant eller duplicerad kod är inte acceptabelt

- Mer (onödig) kod -> mer chanser till fel
- Kod måste hållas i synk, ändringar måste göras på flera ställen.

När ni fått till ett fungerande kodavsnitt …

- .. gör en “code review”!
- Görs något i onödan, eller görs samma sak på flera ställen? Åtgärda!
- Idealet är att allt finns/görs på exakt ett ställe i programmet.

Code smells är tecken på att koden inte är bra!

God Praxis: Arrayer

- Arrayer används framförallt då man inte vill att strukturen skall ändras, vi kan ta bort element men index finns kvar.
- Finns inte det kravet föredra någon samling (t.ex. List)

Ibland tvingas vi använda arrayer p.g.a. att Java's standardmetoder returnerar dylika eller kräver arrayer som parameter.

God Praxis: Metoder

- En metod skall vara expert på en sak!
- Hellre många små specialiserade metoder än några stora
- “one-liners” metoder är helt ok.
- Skilj IO och logik
- Undvik utparametrar
- Namnet innehåller ett verb eller är en ja/nej-fråga (booleska metoder)
- Undvik att returnera null (skicka t.ex. en tom array/List)

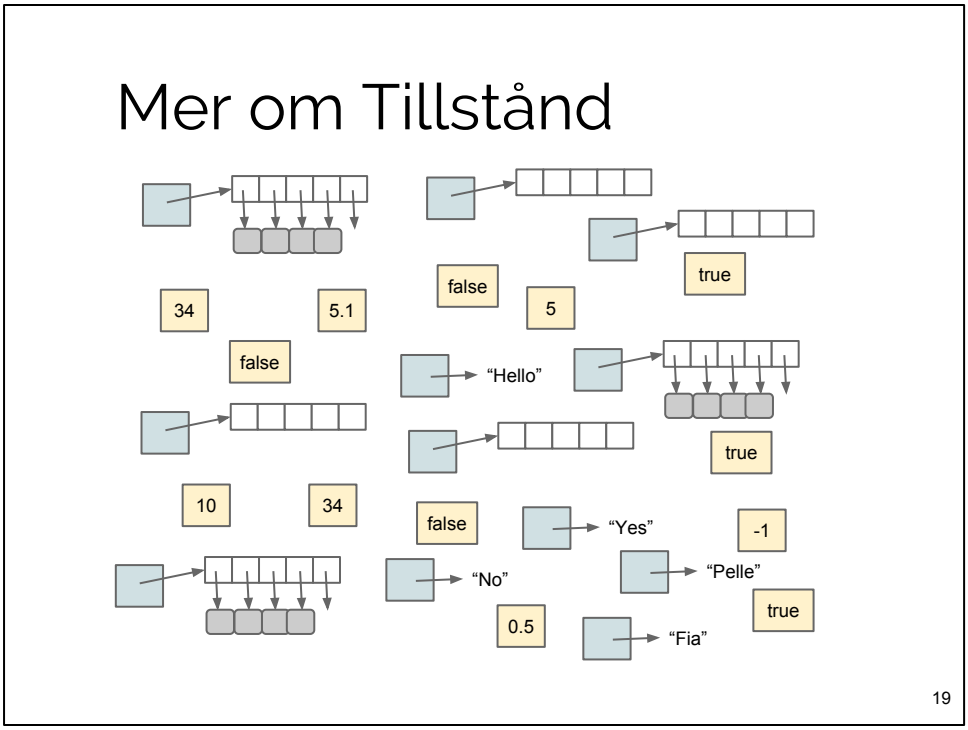
God Praxis: Klasser

18

En klass skall fånga ett koncept

- En klass skall ha ett ansvarsområde
- … på samma sätt som för metoder
- … annars blir de svåra att (åter)använda, felsöka, … det blir rörigt!
- Bättre att kombinera flera (små) tydliga klasser

Bilden: Så här vill vi inte att en klass skall fungera (för mycket olika saker)

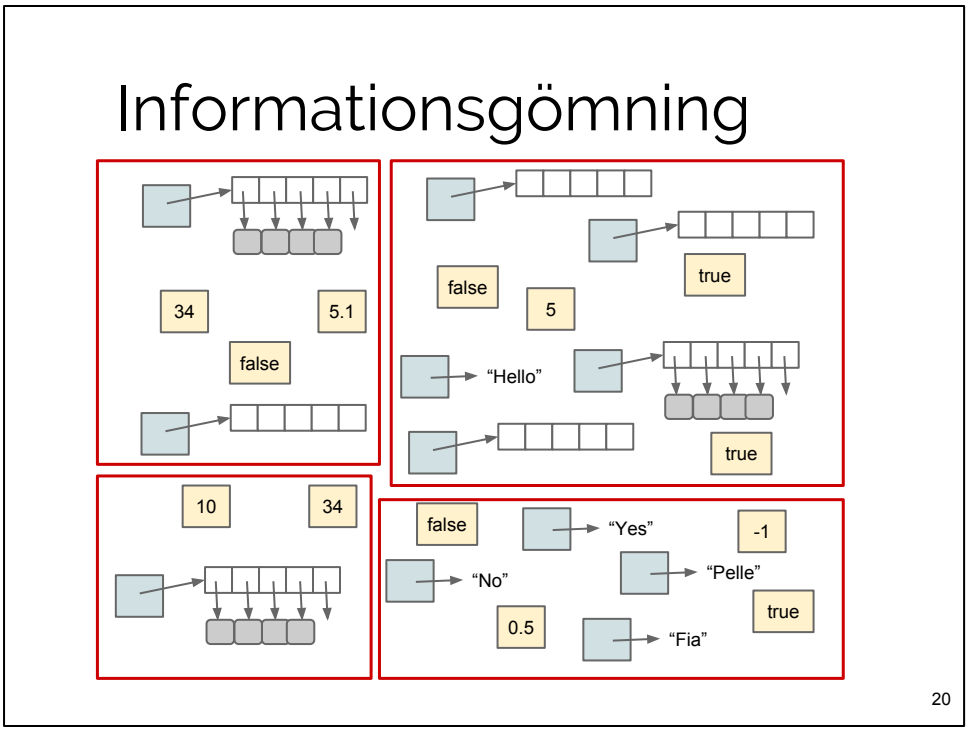


Tillstånd (state) mängden av alla värden för alla instansvariabler i programmet vid en viss tidpunkt under exekveringen

- Lokala variabler räknas ej (de kommer och går …)
- Om allt fungera som tänkt innehåller variablerna korrekta värden, programmet befinner sig i ett giltigt tillstånd … om EJ
- … har vi ett ogiltigt tillstånd. Något är fel
- Att naivt försöka behärska tillståndet övergår mänsklig förmåga …
  - … vi måste utveckla tekniker för detta, forts. …

Ett fundamentalt problem inom imperativ programmering är att behärska tillståndet!

- Innebär t.ex. att vi alltid föredrar lokala variabler (eftersom de inte ingår i tillståndet)
- Vi försöker också konsekvent att minska synlighetsområdet för variabler.



Ett sätt att behärska tillståndet är att dela upp det totala tillståndet i mindre deltillstånd och på så sätt lättare kunna hålla dessa giltiga

- Kallas **informationsgömning** (information hiding), lite svävande terminologi, … men vi säger så)
  - Vi ser till att instansvariabler (i möjligaste mån) bara används inom en viss del av programmet
  - Övriga delar skall inte komma åt
- Dock ingen garanti … lokalt giltiga tillstånd ger inte automatiskt globalt giltigt tillstånd

Exempel : Antag totala tillståndet skall noll

- Deltillstånd A skall alltid vara negativt och deltillstånd B skall alltid vara positivt
- Även om deltillstånd giltiga så inte säkert totalt tillstånd giltigt