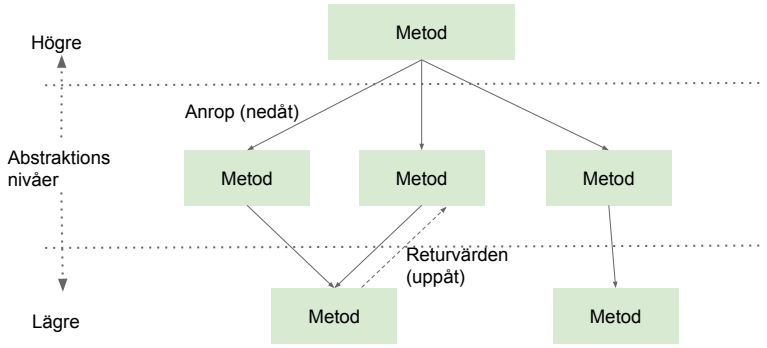
ex4methods

Joachim von Hacht

Konceptuell bild



2

En metod är en avgränsad del av ett program som utför en viss uppgift

- Ger ofta ett visst utvärde/resultat (motsvarar matematisk funktion) …
- … men inte alltid.
- En metod påminner mycket om en matematisk funktion men eftersom alla "funktioner" i Java tillhör ett objekt (mer senare) kallas de metoder (en annan skillnad är att vissa metoder inte har något resultat).

Att använda metoder i ett program ger många fördelar

- Programmet får en struktur, programmet blir greppbart
  - Om ett program överstiger ett par hundra rader börjar det bli ohanterligt ..
  - … genom att strukturera programmet m.h.a. metoder kan vi behärska komplexiteten.
- Vi kan bygga upp programmen bit för bit
  - Man skriver och testar en metod i taget.
- Metoder har namn!
  - Programmet blir lättare att förstå om vi inför begrepp på en högre nivå (säger mer vad det handlar om)
- Metoder kan återanvändas, innebär att vi undviker upprepad kod (redundans)!!
  - Vill vi köra samma kod på flera ställen, anropar vi samma

        - metod
    - Innehållet i en metod kan ändras utan att resten av programmet behöver ändras.

I bilden: Metoder arbetar på olika abstraktionsnivåer:

- Metoderna på låg abstraktionsnivå är mycket detaljerade (använder ofta primitiva typer)
- På nästa nivå är de mer övergripande och åstadkommer mer (arbetar med en större del av problemet).
    - OBS! Metoder på högre nivå anropar de på lägre. Returvärden skickas från lägre nivå till högre.
- Kallas en skiktad design, mer i senare kurser.

Metoddeklaration

```
// Declaration of method named add
// Parameters int a and int b
// Return type int
int add( int a, int b ){   // Head
  return a + b;            // Body
}
```

4

Man skapar en metod m.h.a. en **metoddeklaration**

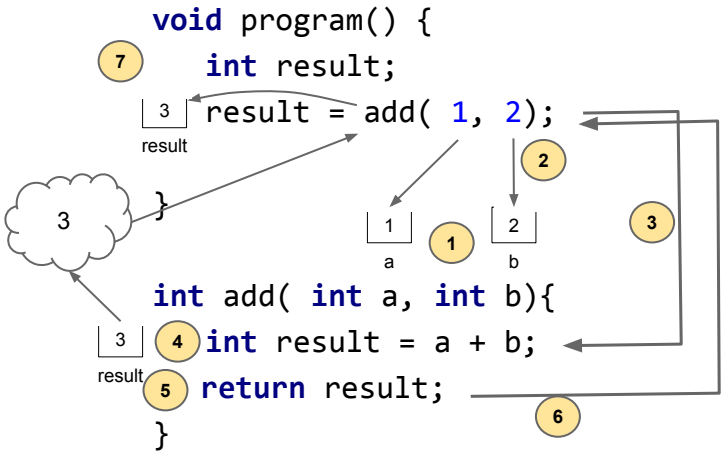
- Första raden i deklarationen kallas metoden **huvud** (head)
  - I huvudet anges (vänster -> höger): Returtyp, namn och en **parameterlista** inom parentes
    - Returtypen anger typen på värdet som metoden returnerar (om något, mer senare …)
    - Namnet väljer vi själva (efter de regler som finns för namn)
      - Namnet skall vara lagom långt och beskriva vad metoden skall göra, ofta är ett verb inblandat
      - Metodnamn inleds med liten bokstav därefter camelCase.
    - I parameterlistan deklareras ett 0-n variabler (parametrarna).
      - Dessa används för att ta emot inkommande data vid anropet (under körning).
      - För varje parameter anges typ och namn (även här väljer vi namn)
- Koden i blocket efter huvudet kallas metodens **kropp**. I kroppen skrivs den kod som skall köras då metoden anropas
- En **return**-sats i kroppen används för att avsluta metoden och skicka tillbaka värdet som står efter return.

- Om det står ett uttryck efter beräknas detta först
- Returtypen i huvudet och typen på uttrycket som returneras måste vara kompatibla, annars typfel
- Om vi anger en returtyp måste vi returnera ett värde annars kompileringsfel
    - Kompilatorn kontrollerar att det garanterat finns en return-sats som kommer att köras
    - Om man t.ex. har en if-sats i metoden måste man ev. ha flera return-satser
- En metod kan bara returnera ett värde!
    - Värdet kan dock var sammansatt t.ex. en array

Program innehåller normalt många metoddeklarationer

- Metoder skrivs i det yttersta blocket (samma nivå som program())
    - Inte metoddeklarationer i metoder (de får inte vara nästlade)!
- I vilken ordning deklarationerna skrivs i programmet spelar ingen roll
  Vi lägger inledningsvis alla metoddeklarationer i slutet av programmet

Metodanrop



Att exekvera (köra) metoden, sker genom att **anropa** den, görs genom att skriva metodens namn och aktuell indata inom en parentes.

- Indatan kallas **argument** (fast i JLS kallas de formella parametrar)
- Argumenten måste matcha parametrarna (antal, position, typkompatibla) annars kompileringsfel.
- Om argumenten är uttryck som behöver beräknas gör detta först (ev. implicita typomvandlingar görs också). Beräknings sker v->h.

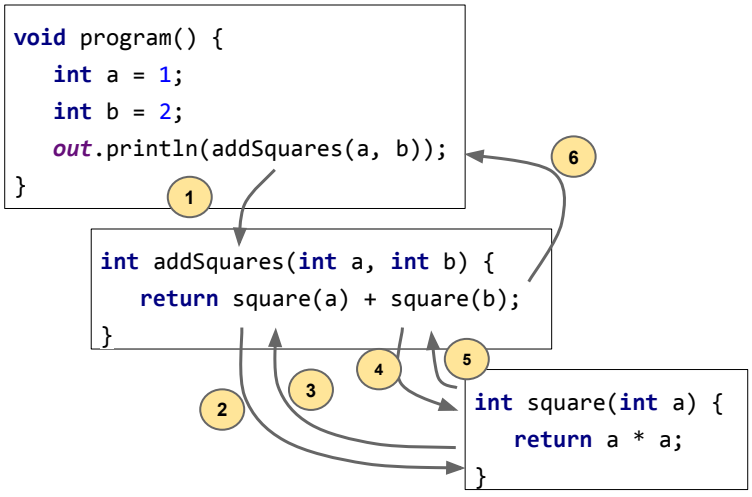
Följande sker vid anropet av metoden (detta skall ni kunna utantill!)

1. Variablerna i metoden (inkl. parametrar) skapas i en del av minnet kallat programmet **anropsstack (call stack).**
2. Argumentenvärden kopieras till metodens nyskapade parametrar (utifrån position).
   - Kallas **värdeanrop (call by value).** Så sker alltid i Java
3. Därefter sker ett hopp, från aktuell (påbörjad) sats till den första satsen i metoden
4. Metoden exekveras sats för sats till en return-sats påträffas
5. Vid return-satsen kopieras returvärdet till en tillfällig lagringsplats, därefter frigörs minnet på anropsstacken, d.v.s. variablerna i metoden försvinner!!!
6. Programmet hoppar tillbaks till den påbörjade satsen (återhopp)
7. Om vi inte tilldelar returvärdet kommer det att försvinna då nästa sats

1. körs
2. I koden i bilden gör vi en tilldelning, vi sparar värdet. Typen på returnerat värde och variabeln måste vara kompatibla, annars typfel

OBS! Att metoder alltid jobbar med kopior av värden..

Metodanrop från Metod



En metod hoppar alltid tillbaks till satsen där den anropades

- Kan ske i flera steg, om en metod anropar en annan

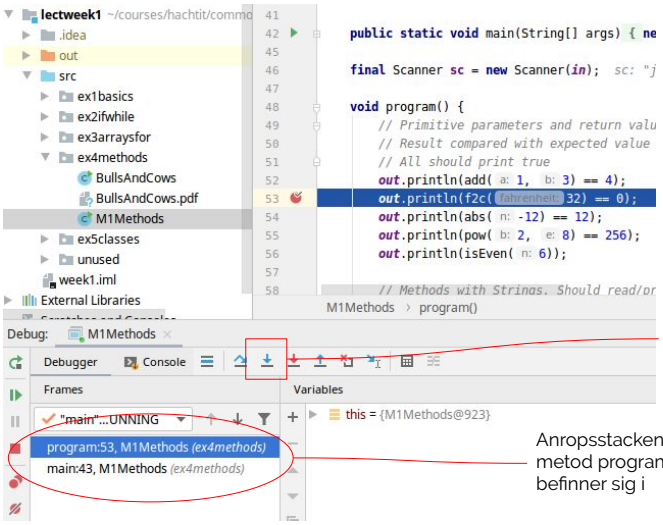
Testning av Metoder

```
void program() {
   // Compare method result to known correct value
   out.println(f2c(32) == 0);     // Should print
   out.println(abs(-12) == 12);   // Should print
   ...
}

int add(int a, int b) {        // Methods to test
   return a + b;
}

double f2c(double fahrenheit) {
   return (fahrenheit - 32) * 5 / 9;
}
```


Vår strategi är följande:

- Så fort vi implementerat en (icke trivial) metod så testar vi den (de i bilden är triviala men bara ett exempel).
- Vi automatiserar testerna genom att skriva ut 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 behåller alltid alla tester, … när som helst kan vi köra dessa igen!
- IO-metoder testar vi, vid behov manuellt, vi provkör helt enkelt
  - IO metoder är (skall vara) enkla.

Avlusa Metoder



7

Det gå utmärkt att avlusa metoder. Gör som vanligt men när metodanropet kommer klicka på stega in (röd fyrkant i bilden).

- Avlusaren hoppar ner till metoden och där kan man stega vidare.

<u>OBS! Att felsöka/debuga i tester är mycket effektivt</u>:

- Vi kan hårdkoda in data
- Vi arbetar med isolerade delar av programmet.

I IntelliJ kan man inspektera anropsstacken

- Anropsstacken används då till att hålla reda på i vilken metod programmet är och vart det skall hoppa då metoden är klar.

Numeriska Parametrar och Returtyper

```
// Convert Fahrenheit to Celsius
double f2c(double fahrenheit){
   return (fahrenheit - 32) * 5 / 9;
}

// Absolute value (NOTE: 2 return statements)
int abs(int n) {
   if (n < 0) {
       return -n;
   }
   return n;
}
```


Används för olika typer av beräkningar.

Nästlade metodanrop

```
out.println(sqrt(pow(3, 2) + pow(4, 2)));
```


Det går att skicka returvärdet från en metod direkt som argument till en annan metod

- Kallas **nästlade anrop** (nested calls)
- Ibland användbart … slipper en del "onödiga" variabler för returvärden
  - Speciellt vid utskrifter
- … dock svårt att felsöka, allt sker i ett enda svep

Parametrar evalueras vänster till höger

- Men koda aldrig så att man blir beroende av detta

Boolesk Returtyp

```
// Boolean method (NOTE: No if statement needed)
boolean isGameOver(int score1, int score2){
    return score1 >= 10 || score2 >= 10
                        && score1 != score2;
}
```


Booleska metoder används för att svara på ja/nej frågor

- Kan vara bara en rad med ett (komplext) boolesk uttryck
- Namnges som en ja/nej-fråga, hasWinner(), isEven(), isLeapYear()
- Användbara, höjer abstraktionsnivån, döljer rörig kod

Sträng som Returtyp

```
final Scanner sc = new Scanner(in);

// Get user name (NOTE: No variable used)
String getName() {
   out.print("Please enter your name > ");
   return sc.nextLine();  // Return result at once
}
```


Här i kombination med IO.

- getName är en IO metod.
- Mer senare.

void-metoder

```
void roundMsg(int result, int computer, int statistic) {
   out.println("Computer choose: " + computer);
   if (result == draw) {
       out.println("A draw");
   } else if (result == humanWin) {
       out.println("You won");
   } else {
       out.println("Computer won");
   }
   out.println("Result " + statistic); // No return!
} // Jump back
```

12

void-metoder

- Man kan ange att en metod inte returnerar ett resultat …
- … görs genom att ange **void** istället för returtyp
- Typiskt funktioner som "bara gör något", skriver ut till skärmen t.ex.
- I övrigt fungerar void-metoder som andra metoder
  - Återhopp sker då sista krullparentesen nås.
- Metoden får inte innehålla en return-sats med ett värde efter
  - Däremot bara return går bra, innebär att man avslutar metoden
    - Man kan alltså avsluta "mitt i" en metod
    - Gäller även för icke-void metoder , undvik (men ok vid vissa tillfällen)!
- Eftersom void-metoder inte returnerar något, representerar de inte något värde
  - … de är inte uttryck (de är satser)
  - Kan inte stå t.ex. vid tilldelning

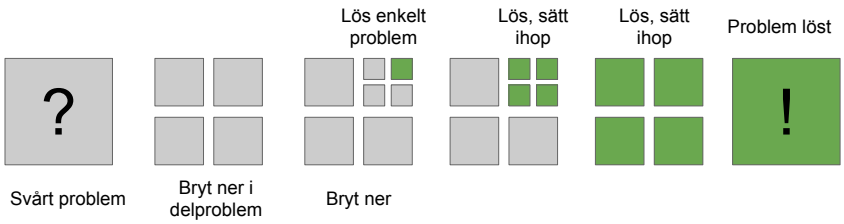
Arbetssätt

13

Ett genomtänkt arbetssätt skall hjälpa er att klara kursens laborationer (...och följande kursers)

- Genom att arbeta på ett visst sätt ökar vi vår förmåga att lösa problem.
- Generellt: Behärska komplexitet.

Bryta ner Problem



14

Normalt är ett problem (program) för stort för att direkt kunna kodas med minsta steget metodiken.

- Vi måste dela upp problemet.

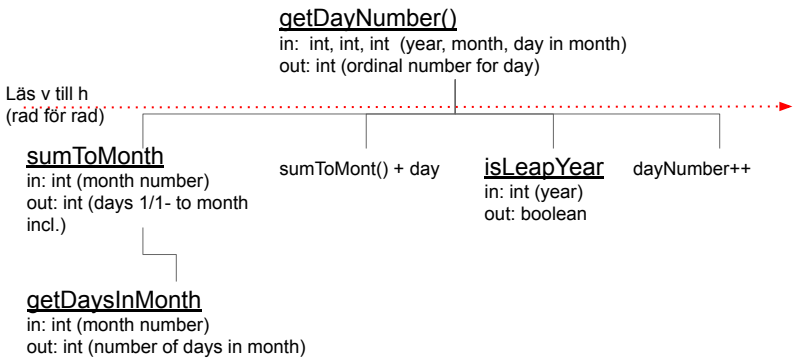
Ett klassiskt angreppssätt om man har ett stort/svårt problem.

- 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

Problem: Givet årtal och datum, beräkna ordningsnumret för dagen detta år (Exempel: 25/4, 2018 är dag 115 detta år)



15

Funktionell nedbrytning (functional decomposition) innebär att man:

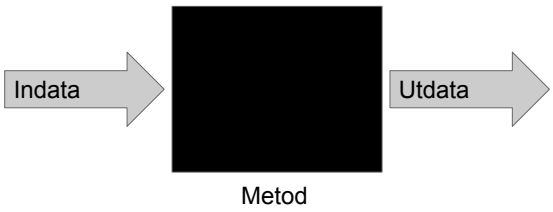
- Antar att man har en metod som löser hela problemet (i bilden: getDayNumber)
- Börjar skriva denna … när man stöter på ett nytt problem antar man att man har en metod som löser detta (sumToMonth, isLeapYear).
- O.s.v… tills metoderna man behöver är triviala t.ex. getDaysInMonth.
  - Implementera de triviala metoderna.
  - Implementera m.h.a. dessa metoderna "högre upp"
  - Genom att kombinera små enkla metoder har man löst ett stort problem. Klart!
- För att illustrera kan man rita ett diagram enligt bilden. Överst ritar man den första metoden man antog, därefter en ny nivå för varje nedbruten method

Funktionell nedbrytning är en top-down strategi. Man börjar med helheten och bryter ner i enklare delar.

Genom att använda funktionell nedbrytning (plus anteckningar/skisser) får vi en struktur på programmet.

- Ha alltid papper och penna uppe!

Funktionell Abstraktion



19

Under arbetet med funktionell nedbrytning söker vi metoder vi behöver.

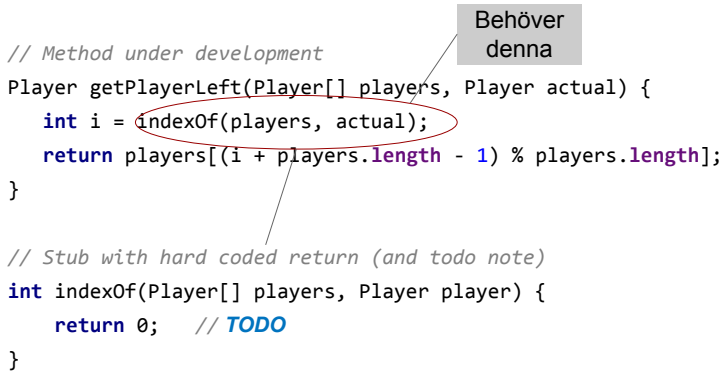
För att underlätta sökandet använder man **funktionell abstraktion**, en tankeform där man bara fokuserar på:

- 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.

Process för att skriva en metod:

1. Namnge metoden (med ett bra namn som säger vad den gör, ett verb är ofta inblandat).
    a. En metod skall vara bra på en sak, kan du inte hitta ett bra namn kan det bero på att metoden gör för mycket. Isf bryt ner den i mindre metoder.
2. Vilken data har du? Ange parametrar.
3. Vad vill du ha? Skriv metodens returtyp m.h.a. detta.
4. Skriv klart metoden m.h.a. minsta steget (och testa, se senare slide)

Metodstubbar



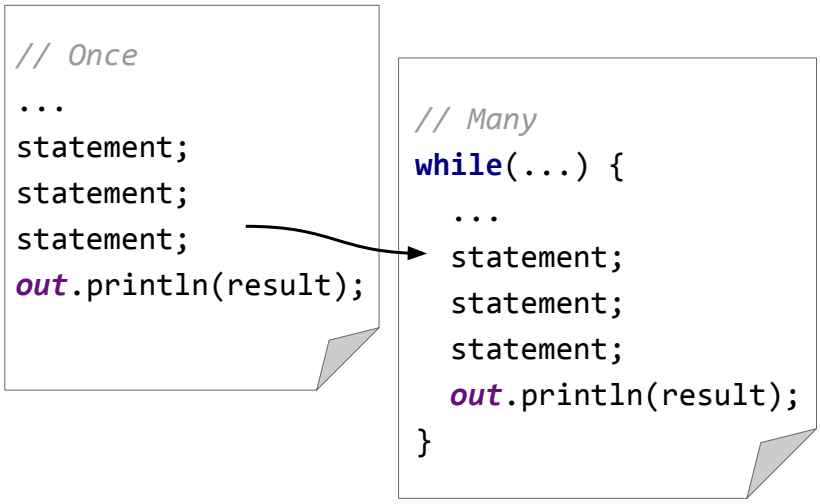
Då man arbetar med funktionell nedbrytning händer det t.ex. att man upptäcker flera metoder samtidigt.

- 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 kan vi få ett körbart program.
  - Vid behov hårdkoda returdata, t.ex. return 0, return null, return "", o.s.v..

En Gång, flera Gånger



En fundamental insikt är att …

- 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!

Ansvar

19

En metod skall ha ett och endast ett ansvar

- Skall göra en sak.
- Gör det möjligt att kombinera ihop flera specialiserade metoder till hela lösningar.
- Gör det möjligt att återanvända.

Variabler skall också ha ett ansvar (inte återanvända variabler för olika data)