Inledande om Arrayer, for-satsen och Metoder

Vecka 2, Bildserie 1

Innehåll

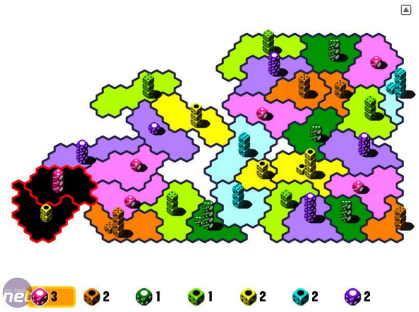
- Arrayer (del 1)
- for-satsen
- Färdiga metoder
- Java-dokumentation
- Egna metoder
- Metoddeklaration
- Metodanrop
- Lokala variabler
- Booleska metoder
- void-metoder
- Logik och IO-metoder

Mål vecka 2

Kunna använda arrayer och for-satsen
Kunna:

- Använda färdiga metoder
- Skriva egna metoder
- Strukturera lösningen till ett problem m.h.a. metoder

Veckans Produkt



```
Welcome to Dice Wars "lite!"

pelle:2--fia:3--lisa:2
     \     |      |
fia:3--lisa:1--pelle:3
   |       |    /  |
pelle:1--fia:1--lisa:3

--------------------------------
(pelle) Action > a
(pelle) Select from > 5
(pelle) Select to > 4
pelle:11 lisa:6

pelle:2--fia:3--lisa:2
     \     |      |
fia:3--pelle:2--pelle:1
   |       |    /  |
pelle:1--fia:1--lisa:3
```

Vi gör en egen textbaserad version av spelet Dice Wars.

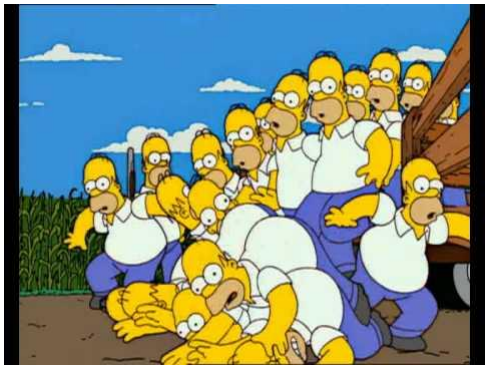
Att Läsa i Boken

- 4.1-4.2
- 5.4-5.5
- 5.9
- 7.1
- 7.2.1-7.2.5
- 6.1-6.6
- 6.8-6.9
- 6.11

OBS! Att jag använder lite annorlunda stil

- Inledningsvis behövs aldrig ordet static eller public, bortse från dessa.

Många av Samma



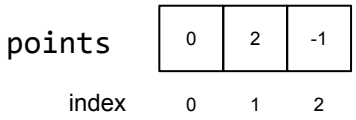
Literaler och variabler kan användas för enstaka värden.

- Men vad händer om vi behöver många …
- … 10 heltalsvariabler, eller 100, eller 100000 …. ?
- Orimligt att deklarera dessa en och en.
- Lösning: Om vi behöver många variabler av samma typ kan vi använda en array …

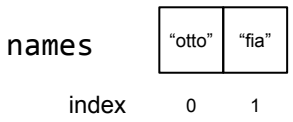
Arrayer

Deklaration och initiering

```
int[] points = {0, 2, -1};
```




```
String[] names = {“Otto”, “Fia”};
```



En **array** är ett antal variabler med ett gemensamt namn

- Jämför: Namn på varje hus eller (gatunamn + nummer) för alla hus

För att använda en array måste den deklareras och initieras.

- Görs m.h.a. en deklaration där man anger namnet och [ ] efter typen
  - Typen med [ ] gäller för hela array:en (vi säger t.ex. int-array för typen)
  - Typen utan [ ] gäller för de enskilda variablerna
    - Alla variabler har samma typ
- Vi initierar array:er direkt genom att ange värdena för de enskilda variablerna i en lista.
  - Variablerna i arrayen skapas och initieras med värden från listan i skriven ordning (från vänster till höger)
- Array:en kommer att innehålla lika många variabler som värden vi angav
- Varje enskild variabel har ett index
  - Första index är 0 och sista (längd-1)
- En array kan inte ändra storlek efter det att vi initierat den.

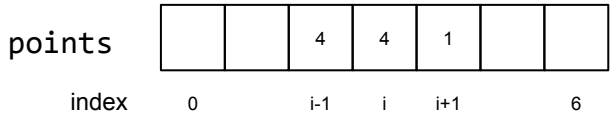
Indexering

```
int[] points = {0, 0, 0};

points[0] = 4;                  // { 4, 0, 0 }
points[2] = 1;                  // { 4, 0, 1 }
points[1] = points[2];          // { 4, 1, 1 }
points[0] == points[2];         // false
points[0] > points[1];          // true

points[6] = 3;                  // Exception
points[-1] = 3;                 // Exception
```

Indexering innebär att komma åt de enskilda variablerna, **elementen**, i en array

- Indexering skrivs: *array-namn* [*index*]
  - [ ] kallas indexeringsoperatorn
- Index måste var ett heltalsuttryck (variabler går bra).
- Vi måste själva se till att index inte hamnar utanför array:en, …
  - … om utanför så undantag vid körning ett **exekveringsfel** (IndexOutOfBoundsException)

Indexering, forts




```
int[] points = {0, 0, 0, 0, 0, 0, 0};

int i = 3;
points[i] = 4;            // { 0,0,0,4,0,0,0 }
points[i+1] = 1;          // { 0,0,0,4,1,0,0 }
points[i-1] = points[i]; // { 0,0,4,4,1,0,0 }
```

En array har en struktur, det finns första/sista, före/efter, vänster/höger

- Ger oss möjlighet att referera till “värdet till höger/vänster”, användbart!
  - En indirekt referens: “grannen till”, “tre efter”, “ en före”, …
  - Ny abstraktionsnivå!
- Givet ett index i kommer vi åt
  - variabeln till vänster (före) med i-1
  - variabeln till höger (efter) med i+1
  - i-1 eller i+1 får inte hamna utanför array:en, om så: undantag (som tidigare).

Längden av en Array

```
int[] points = { 0, 0, 0, 0, 0 };
out.println(points.length)   // 5

int i = 0 ;
while( i < points.length ){
   …  points[i];    // Do something
   i++;
}
```

Man kan komma åt längden av en array på ett fördefinierat sätt (inbyggt i språket)

- Genom att skriva: *array-namn*.length

Traversering

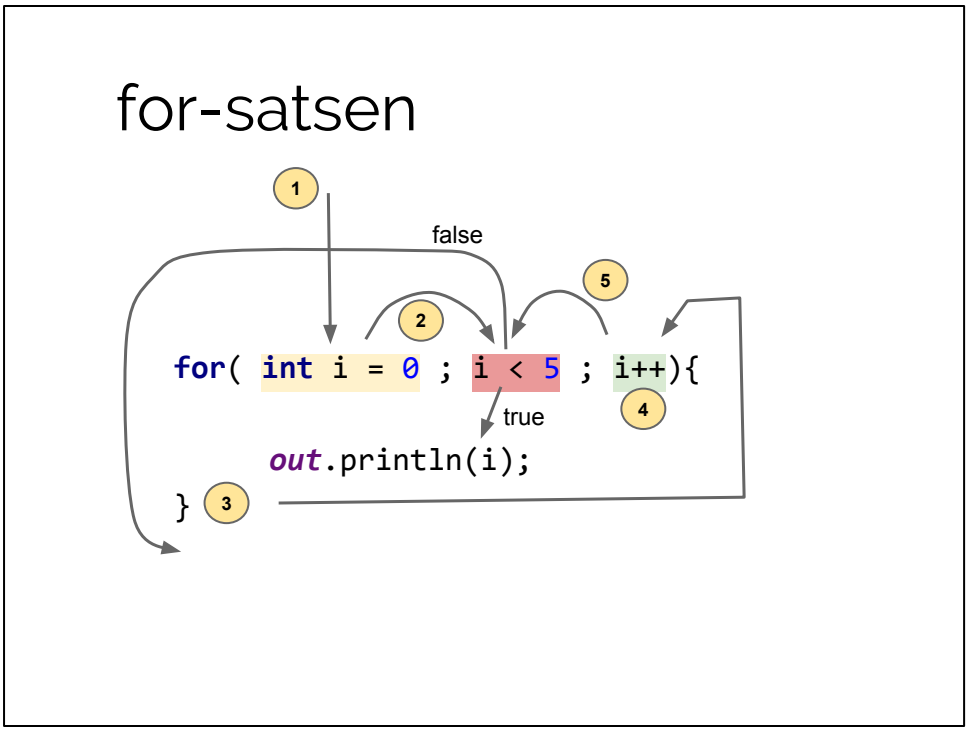
- Ofta vill man **traversera** (genomlöpa) en array d.v.s. komma åt alla variabler i tur och ordning
  - Från vänster till höger eller tvärt om.
- Traversering görs med en while-sats eller vanligare en for-sats (kommer strax)
- Som villkor använder man: i < array.length (vänster till höger)
  - D.v.s. i skall vara strikt mindre än längden eftersom sista index är ett mindre än längden
- eller i >= 0 (höger till vänster), större eller lika med eftersom första index är 0.

Utskrift av Array:er

```
int[] points = { 1, 2, 3, 4, 5 };
int i = 0 ;
while( i < points.length ){
    out.println(points[i] + " "); // 1 2 3 4 5
}

// Simpler "built in" [ 1, 2, 3, 4, 5 ]
String s = Arrays.toString(points);
out.println(s);
```

Man kan inte använda out.println() direkt

- out.println(points) skulle ge en "konstig" utskrift typ: [I@7f31245a
- För att skriva ut en array måste man antingen använda en loop och skriva ut värdena ett i taget…
- .. eller det inbyggda sättet Arrays.toString(), omvandlar arrayen till en sträng som sedan kan skrivas ut.



I Java finns flera typer av loopar

- Alla är logiskt utbytbara, vi kan klara oss med t.ex. bara while
- while-satsen betecknar konceptet upprepa-tills … vi vet inte på förhand hur många gånger vi skall upprepa
- Ibland vet man på förhand hur många gånger man skall upprepa
  - I dessa fall använder vi for-satsen (for-loopen)
- Dessutom: for-satsen samlar all information om loopen på en enda rad (många tycker detta är bra, överskådligt …)

Analys av kod (for-loopen består av 3 delar inom parentesen och ett efterföljande block)

1. Räknaren (variabeln i) för antal varv deklareras och initieras (sker bara första gången)
   Synlighetsområdet för räknaren är bara i for-loopen (i parentesen och blocket efter)
2. Villkoret evalueras
   a. Om Sant
      - Blocket efter for exekveras
      - När slutparentes i blocket nås sker ett hopp till sista delen av parentesen (3)
      - Räknaren ökas/minskas (4)
      - Därefter evalueras villkoret igen (5)

a. Om falsk
    - Hela blocket hoppas över

OBS!

- VARNING: Skall vara ";" mellan delarna i parentesen annars konstiga fel
- Som tidigare: Undvik att förändra räknaren i loopen (den skall bara räknas upp/ner i sista delen

for och Array:er

```
int[] arr = { 1,2,3,4,5,6 };

for( int i = 0 ; i < arr.length ; i++){
   out.print(arr[i]);    //123456
}

for( int i = arr.lenth-1 ; i >= 0 ; i--){
   out.print(arr[i]);   //654321
}
```

Vanligen används en for-loop för att traversera en array

- Eftersom vi vet på förhand hur många varv vi skall köra (nämligen length)
- Man använder som förut, i < array.length (strikt mindre) eller i >= 0, som villkor

Färdiga Metoder

```
import static java.lang.Math.*; // Must have

double d = sqrt(25); // 5.0
d = pow(5, 2);       // 25.0
d = floor(4.6);      // 4.0
d = ceil(4.4);       // 5.0
int i = floor(4.4);  //Type error
int j = ...
d = pow(6, 2*j);    // Type compatible
d = ceil(pow(arr[j], j));
```

I Java finns ett mycket stort antal färdiga funktioner

- Funktioner kallas i Java för **metoder** (method)

Behöver vi t.ex. numeriska metoder i vårt program anger vi: import static java.lang.Math.*;

- Vi får då tillgång till metoder för kvadratrot, logaritm, trigonometri, m.fl.
- Finns även färdiga konstanter för **π** (PI) och e (E), mer om konstanter senare …

Om man i programmet vill **anropa** (call, invoke) en metod skriver man namnet och anger **argumentet(en)/parametrar(na)** i parentesen

- Värdena i parentesen (som skall skickas till metoden) kallas för **aktuella parametrar**
- Parametrar måste vara typkompatibla med det metoden förväntar sig annars typfel.
    - **Parametertyper** för metoderna i bilden är double (för alla parametrar)
        - Men anrop med int går bra, det är typkompatibla
    - Parametrarna måste stå i rätt ordning (t.ex. bas,exponent inte exponent,bas)
    - Om parametern är ett uttryck beräknas detta först

        - Om flera uttryck beräknas de från vänster till höger.
- Anropet ger (i fallet ovan) ett resultat, ett **returvärde** (som har en typ, **returtypen**, alla värden har en typ!)
    - … innebär att en metod har en typ, typen på returvärdet (annars fungerar inte typsystemet)!
        - Ett **metodanrop** (enligt ovan, mer senare…) är alltså ett uttryck
        - Alla metoder ovan har double som returtyp
        - Använder vi en int-variabel visar IntelliJ på ett typfel
- Vilka parametrar man skall skicka och vilken typen på resultatet man får finns dokumenterat här

Nästlade metodanrop

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

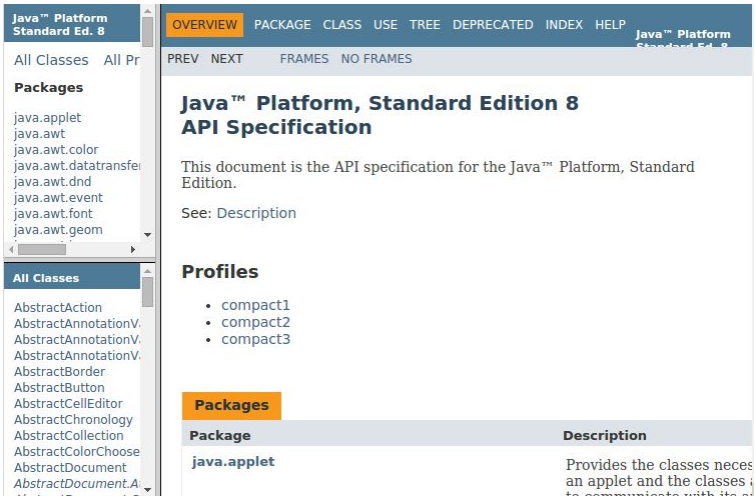
Det går att skicka returvärdet från en metod direkt som aktuell parameter till en annan metod

- 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→ höger

- Men koda aldrig så att man blir beroende av detta

Java-Dokumentation



Hur vet man vad som finns färdigt i Java?

- Allt "färdigt" finns i Javas **standardbibliotek** kallas även API:er
  - Följer med när du laddar ner Java
  - Javas standardbibliotek innehåller 10 000-tals metoder m.m....
- Standardbiblioteken är dokumenterade
  - Innebär att man samlar beskrivningar av metoder:
  - Man anger: Namn, vad metoden gör, parametrar och returvärden samt typer för dessa
- Dokumentationen (kallas ofta Javadoc)

En svårighet med moderna språk är att de innehåller enorma standardbibliotek

- Förutom själva språket måste man lära sig hitta bland allt färdigt.
- I denna kurs använd bara en mycket liten del … (inga större problem, ABSOLUT inget man behöver lära sig utantill)

Modern programutveckling innebär

- att man söker igenom Web:en efter bibliotek (standard eller andra s.k. tredjeparts-bibliotek) med lämplig kod för uppgiften
- att man därefter sammanfogar den funna koden med "så lite som möjligt" av egen kod

Egna Metoder

Om vi saknar någon metod kan vi skapa en egen som utför det vi vill.

Att använda metoder ger många fördelar

- 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!!
  - Vill vi köra samma kod på flera ställen, anropar vi samma metod
- 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.

Metoddeklaration

```
void program() {
   int result;
   result = add( 1, 2 ); // Call
}

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

Metoddeklarationer

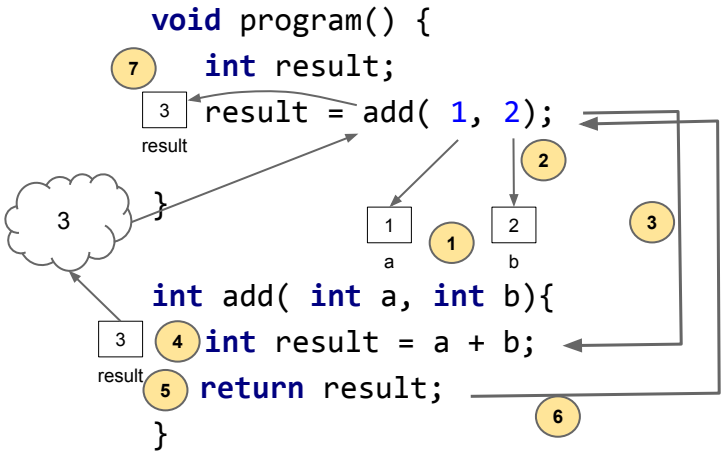
- Man skapar en metod m.h.a. en **metoddeklaration**
- Första raden i deklarationen kallas metoden **huvud** (head)
    - I huvudet anges (vänster -> höger): Returtyp, namn och en **parameterlista** inom parentes
        - Returtypen anger typen på värdet som metoden returnerar
        - Namnet väljer 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.
            - Parameterlistan innehåller metodens **formella parametrar** (de beskriver formellt vad som måste skickas som indata, vad metoden behöver för att göra sitt jobb)
            - I parameterlistan skrivs noll eller flera parametrar åtskilda av komma
            - 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 stämma, 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 måste man ev. ha flera return-satser
        - … undantag se void-metoder senare

Program innehåller normalt flera metoddeklarationer

- I vilken ordning deklarationerna skrivs i programmet spelar ingen roll (de får inte vara nästlade)
- Vi lägger inledningsvis alla metoddeklarationer i slutet av programmet

Metodanrop



Anrop görs genom att skriva metodens namn och aktuell indata inom parentes efter (som tidigare)

- Om de aktuella parametrar, inte matchar de formella parametrar (antal, typkompatibla utifrån position) så kompileringsfel.
- Om parametrarna är uttryck som behöver beräknas, gör detta först (ev. implicita typomvandlingar görs också)

Om allt är ok sker följande vid anropet av metoden

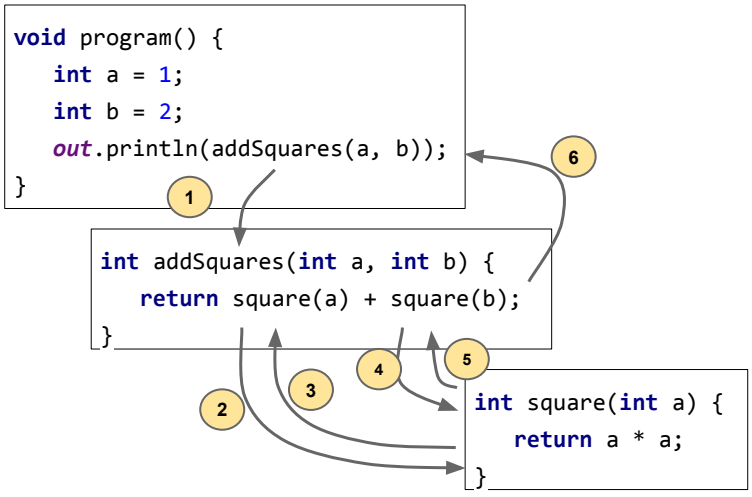
1. Variablerna i metoden (inkl. formella parametrar) skapas i en del av minnet kallat programmet **anropsstack (call stack)**
2. De aktuella parametrarnas värden kopieras till metodens formella (utifrån position). Kallas **värdeanrop (call by value).** Java använder enbart värdeanrop
3. Det sker ett 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 lagringplats, därefter frigörs minnet på anropsstacken, d.v.s. de lokala variablerna försvinner!!!
6. Programmet hoppar tillbaks till den påbörjade satsen (återhopp)
7. Om vi inte tilldelar returvärdet kommer det att förvinna då nästa sats körs

a. I koden i bilden gör vi en tilldelning, vi sparar värdet.
b. Typen på returnerat värde och variabel måste vara kompatibla, annars typfel

OBS! Att metoder alltid jobbar med kopior av värden, d.v.s. inga variabler utanför metoden påverkas

- Bra, så slipper vi leta fel överallt, men … mer kommer …

Metodanrop från Metod



En metod hoppar alltid tillbaks till satsen där den anropades

- Kan ske i flera steg, om en metod anropar an annan
- Anropsstacken används då till att hålla reda på i vilken metod programmet är och vart det skall hoppa då metoden är klar
  - I IntelliJ kan man inspektera exakt vad som händer men anropsstacken (i debuggern)

Stack Overflow

```
void program() {
   program();  // Oh, ooh
}
```

```
/home/hajo/devel/jdk1.8.0/bin/java ...
Exception in thread "main" java.lang.StackOverflowError
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
    at samples.basics.ForLoop.program(ForLoop.java:14)
```

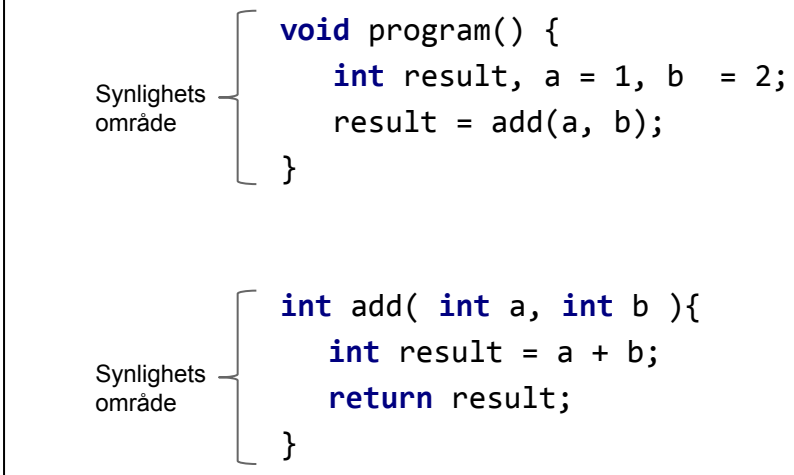
Vid varje anrop skapas lokala variabler på anropsstacken

- Vad händer om en metod anropar sig självt?
- … om inget händer som stoppar metoden får vi StackOverflowException
- … allt minne för anropsstacken är fyllt.

I vissa fall kan det vara en elegant metod att lösa problem genom att låta en metod anropa sig själv

- Men då måste parametrarna ändras inför för varje anrop
  - M.h.a. parametrarna kan vi konstruera ett villkor som stoppar anropen
- Kallas **rekursion**, mer senare ….

Lokala Variabler



Variabler deklarerade i metoder kallas **lokala variabler**

- Vi räknar även parametrar som lokala variabler
- Lokala variabler är bara synliga i metoden de är deklarerade i (metodkroppen är ett block)
- Lokala variabler existerar bara under tiden metoden körs (som vi såg nyss)
- Lokala variabler måste ges ett värde (initieras eller tilldelas) innan de används, om ej kompileringsfel
- Tills vidare tillåter vi bara lokala variabler i våra program! (d.v.s. inga variabeldeklarationer utanför metoder)

Bilden:

- "result" i koden ovan syftar på två olika variabler med samma namn, den ena i program() den andra i add()
- På samma sätt med a och b.
- I exemplet finns totalt 6 variabler (2 st a, 2st b, 2st result)
  - Vilka existerar under olika tidpunkter.

Anm: I program() ovan deklarerar vi flera variabler på samma rad

- Möjligt att göra men inte så bra, bättre med en/rad (görs av utrymmesskäl här)

Booleska Metoder

```
// Boolean method
boolean isGameOver(int score1, int score2){
    return score1 >= 10 || score2 >= 10
                        && score1 != score2;
}
```

Booleska metoder används för att svara på ja/nej frågor

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

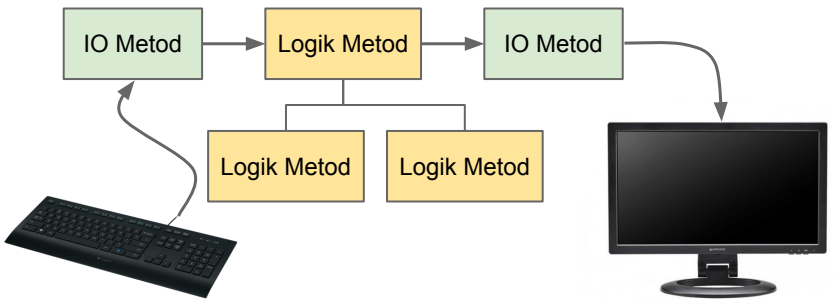
void-metoder

```
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
```

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.
- 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)!
- I övrigt fungerar void-metoder som anda metoder
  - Återhopp sker då sista krullparentesen nås (om ingen return påträffas innan).
- 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

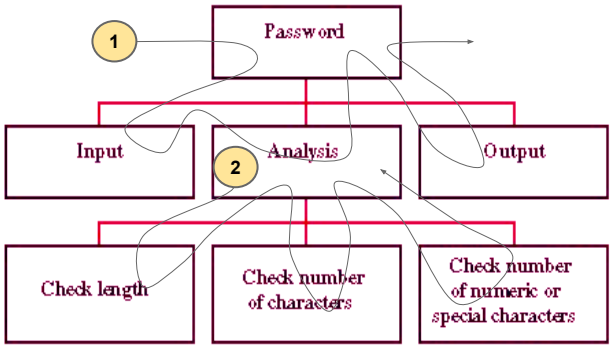
Logik- och IO-Metoder



Precis som tidigare skiljer vi på IO och logik

- En logik-metod använder aldrig IO
  - Aldrig strömmarna in och out
  - Tilldelningar, beräkningar, logiskt flöde sköts här.
- En IO-metod utför ingen programlogik
  - Finns det if och while här är dessa bara till för att utskriften skall bli på ett visst sätt, ingen annat
  - Metoden använder strömmar för att läsa in eller mata ut.
  - IO metoder för utmatning är normalt void.
- Vissa metoder som ansvarar för en större del av ett program (högre upp i hierarkin) använder ibland både IO och logik.

Metodik



För att lösa ett programmeringsproblem (och strukturera lösning) kan man använda **funktionell nedbrytning**

Innebär att man

- Antar att man har en metod som löser hela problemet (Password)
- Börjar skriva denna … när man stöter på ett 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.
- Klart!
- Funktionell nedbrytning är en **top-down** strategi. Man bryter ner problemet i enklare delproblem.

Under arbetet 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!

Programmering

```
Input a year >2016
Input a month number >4
Input a day number >13
Day number for 13/4 in 2016 is: 104
2016 is a leap year
```

Hur skriver man programmet?

- Programmet skall byggas upp med metoder!
- Använd funktionell nedbrytning

Programmering

```
Input coeffs. for grade 4 polynomial (space between) >
3 2 1 -4 12
Derivate is: 12x^3 + 6x^2 + 2x^1 + -4
```

(3 2 1 -4 12 representerar polynomet $3x^4 + 2x^3 + x^2 - 4x + 12$)

Hur skriver man programmet?

- TIPS: for och array

God Praxis

- En metod skall vara expert på en sak!
- Hellre många små specialiserad metoder än några stora
- “one-liners” metoder är helt ok.
- Skilj IO och logik
- Använd funktionell nedbrytning och funktionell abstraktion

Mallen igen …

```
public class MyProgram {
     public static void main(String[] args) {
        new MyProgram().program(); // Call program
    }

    void program() {
        …
    }

    …   // More methods here
}
```

En ny analys av mallen vi använder för våra program.

- Mallen innehåller två metoder
- **main**, som saknar returvärde och tar en array med strängar som argument
    - main måste ha ett metodhuvud exakt som i koden, mer senare …
    - public static hoppar vi över … men
    - … main är uppenbart en metod (parenteserna efter namnet visar detta!)
    - void står före namnet alltså inget returvärde
- Main är den första metod som anropas då ett Java-program startar
    - Detta är förbestämt
- program(), en metod som saknar returvärde och är parameterlös
    - program() anrops från main, … (vi skapar ett program först med new MyProgram(), därefter sker anropet. mer senare …)
    - D.v.s. våra program kommer att starta i metoden program().