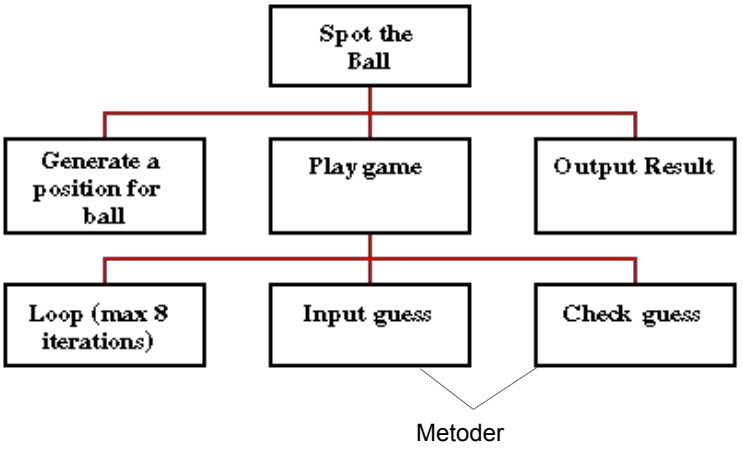
Metoder

TDA548/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 (motsvarar matematisk funktion) …
- … men inte alltid.

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

För att åstadkomma strukturen se: Bildserie Programkonstruktion

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
}
```

3

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

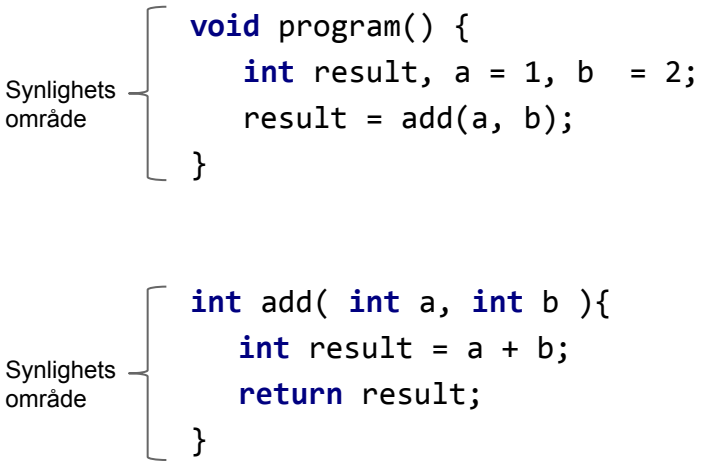
- Första raden i deklarationen kallas metoden **huvud** (head)
  - I huvudet anges (vänster -> höger): Returtyp, namn och en **parameterlista** inom parentes
    - Returtypen anger typen på värdet som metoden returnerar
    - Namnet väljer vi själva (efter de regler som finns för namn)
      - Namnet 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 **parametrar** (de beskriver 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 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

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

Lokala Variabler



5

Variabler deklarerade i metoder kallas **lokala variabler**

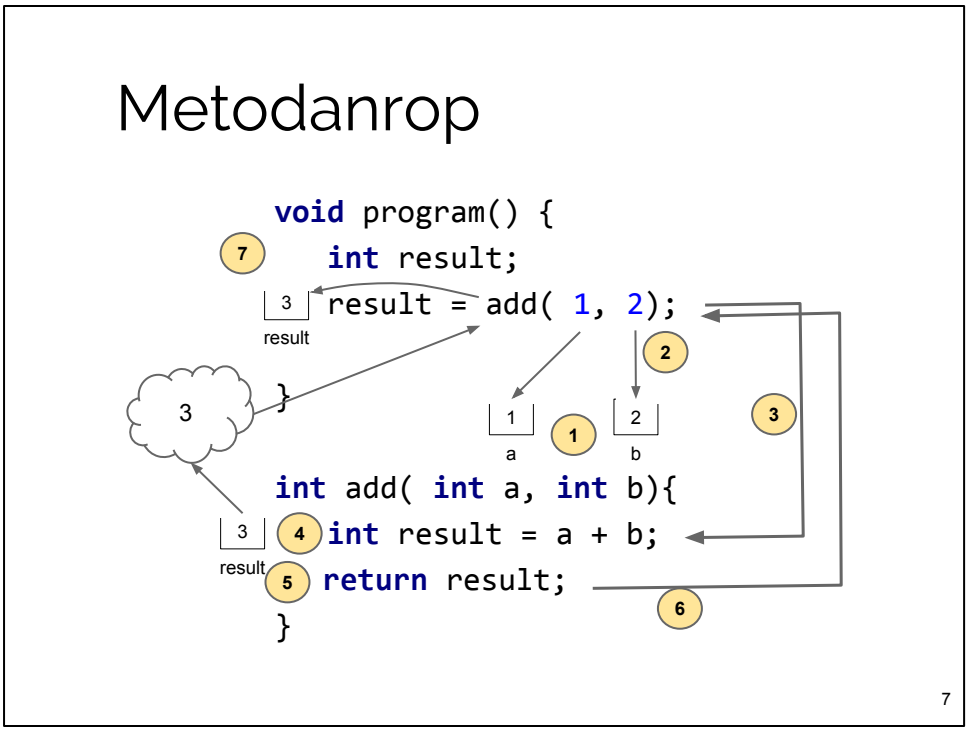
- Vi räknar även parametrar som lokala variabler
- Synlighetsområdet är metodkroppen (ett block)
- Lokala variabler har en livslängd, mer strax …
- Lokala variabler måste ges ett värde (initieras eller tilldelas) innan de används, om ej kompileringsfel
- Tills vidare tillåter vi bara lokala variabler i våra program! (d.v.s. inga variabeldeklarationer utanför metoder)
    - Generella gäller: Föredra lokala variabler!

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.
- Eftersom de finns i olika synlighetsområden kan vi använda samma namn!
- I exemplet finns totalt 6 variabler (2 st a, 2 st b och 2 st result)

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)



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

- Indatan kallas **argument**
- Argumenten måste matcha de formella parametrar (antal, position, typkompatibla) annars kompileringsfel.
- Om argumenten är uttryck som behöver beräknas gör detta först (ev. implicita typomvandlingar görs också)

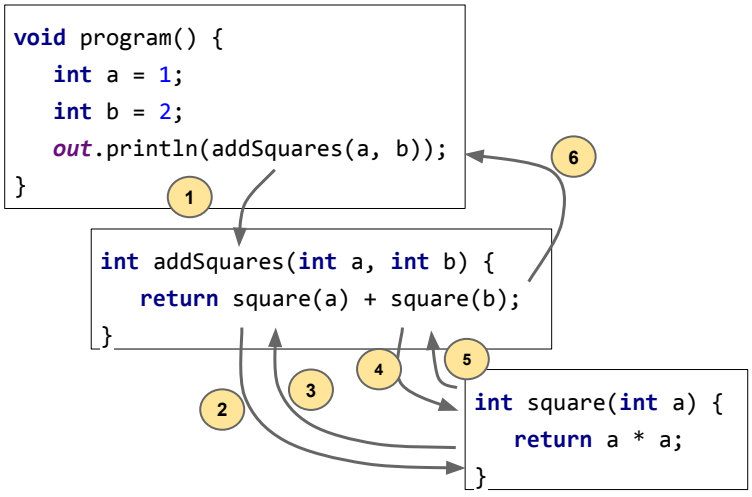
Följande sker vid anropet av metoden

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

1. 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
- 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)

Numeriska Parametrar och Returtyper

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

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


Används för olika typer av beräkningar.

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

- Ofta 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 command
String getName() {
   out.print("Please enter your name > ");
   return sc.nextLine();
}
```


Här i kombination med IO.

- Mer senare.

Array som Parameter*)

```
// Find max value in array
int max(int[] arr) {
   int m = arr[0];
   for (int i = 1; i < arr.length; i++) {
       if (arr[i] > m) {
           m = arr[i];
       }
   }
   return m;
}
```

*) Detaljer senare

Objekt som Parameter*)

```
// Print complete dog
void printDog(Dog dog) {
   out.print("Name: " + dog.name);
   out.println(" Age:" + dog.age);
}

// Class declaration
class Dog {
   String name;
   int age;
}
```

*) Detaljer senare

Array med Objekt som Parameter

```
Dog findOldest( Dog[] dogs){
   int index = 0;
   int maxAge = dogs[index].age;
   for( int i = 0 ; i < dogs.length ; i++){
       if( dogs[i].age > maxAge){
           index = i;
           maxAge = dogs[i].age;
       }
   }
   return dogs[index];
}
```


Kan skicka komplicerade argument

- Här en parameter för en array med Dog-objekt

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
```

13

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

Nästlade metodanrop

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

14

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

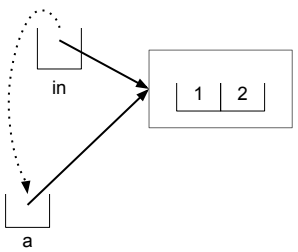
Parametrar evalueras vänster till höger

- Men koda aldrig så att man blir beroende av detta

Mer om Array-parametrar

```
void program() {
   int[] in = { 1,2 };
   zero(in);
   // in is now [ 0, 0 ]
}

void zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
      a[i] = 0;
   }
}
```



Om en metod har en array-parametrar sker precis samma sak som vanligt men …

- … argumentet som kopieras är en referens!
- Parametern kommer därför att peka på samma objekt som argumentet!
- Metoden kan ändra på ett objekt utanför sig självt (variabeln "in" kan vi däremot aldrig ändra)

Summa:

- Samma variabelnamn inom olika synlighetsområden syftar på på olika variabler
- Olika variabelnamn (ev inom olika synlighetsområden) kan syfta på samma sak, om referenser är inblandade

I koden ovan kommer array:en som variabeln in refererar att vara förändrad efter metodanropet

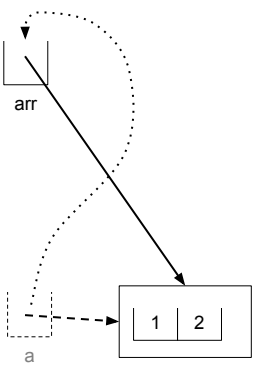
- Kallas **utparameter**
- Innebär en viss risk eftersom det kan vara svårt att inse att en metod har ändrat i objektet (eftersom den är void)

Mer om Array som Returtyp (1)

```
void program() {
   int[] arr = getArray();

}


int[] getArray(){
    int[] a = { 1, 2 };
    return a;
}  // a gone here
```



Om man returnerar en array sker samma sak som vid ett vanligt metodanrop

- Dock är returvärdet en referens

Vi kan skapa en array i en metod och skicka som returvärde.

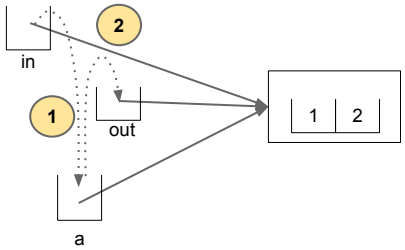
- Den lokala variabeln a försvinner! …
- … men inte array-objektet
- För att kunna komma åt array-objektet måste vi spara den returnerade referensen

Mer om Array som Returtyp (2)

```
void program() {
   int[] in = { 1,2 };
   int[] out = zero(in);

}

int[] zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
       a[i] = 0;
    }
    return a;  // Return parameter!
}
```



Här returnerar vi samma referens som vi skickar in.

- Kan i vissa fall ge smidigare kod.

Överlagrade Metoder

```
// Overloaded (from Math API)
double max( double d1, double d2){...}
float max( float d1, float d2){...}
int max( int d1, int d2){...}
long max( long d1, long d2){...}
```

18

Metoder inom samma synlighetsområde får ha samma namn …

- … men då måste parameterlistorna skilja sig åt!
  - Returtypen spelar ingen roll, bara parametrarna räknas.
- Kallas att metoderna är **överlagrade** (overloaded)
  - Innebär att redan vid kompileringen väljs (utifrån bästa matchning) vilken metod som skall anropas vi körningen (vissa andra metoder bestäms under körning mer i senare kurser …).
  - Vilken metoden som väljs beror alltså på parametrarna.
  - Jämför +-operatorn vars beteende beror på operanderna!
- Tanken med överlagrade metoder är att man skall slippa hitta på nya namn på metoder som gör "samma sak" men med olika antal eller parametertyper.
- Metoder som gör olika saker skall inte ges samma namn

Generiska Metoder

```
// T is the type variable NOTE: Method doesn't use the element objects
<T> int find(T[] arr, T value) {
   for (int i = 0; i < arr.length; i++) {
       if (arr[i] == value) {    // Reference identity!
           return i;
       }
   }
   return -1;
}

// Will convert elements to wrapper type
Integer[] ints = {1, 4, 6, 2, 7, 0, -1};
String[] strs = {"aaa", "xxx", "fff", "ccc", "ddd"};
out.println(find(strs, "aaa") == 0);
out.println(find(ints, 7) == 4);     // Parameters boxed to Integer
```

19

Ibland blir det klumpigt med överlagring, ett alternativ är Generiska metoder.

Generisk metoder kan ta vilken referenstyp som helst som parameter och returtyp

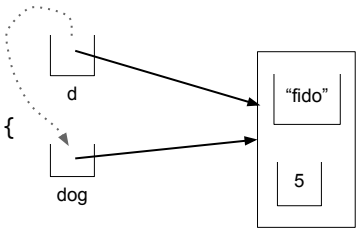
- Haken är att man inte kan göra något med själva objektet, man kan bara arbeta med referenserna t.ex. flytta runt dessa på olika sätt.
- Man anger generisk typ m.h.a. av en typparameter inom vinkelparenteser, normalt kallad T, först av allt i metodhuvudet
- Generiska metoder kan inte ta primitiva typer
    - För att lösa detta kan man använda omslagstyper.

Mer om Objekt som Parameter

```
Dog d = new Dog();
incAge(d);

void incAge( Dog dog ){
   dog.age++;
}

class Dog {
   String name;
   int age;
}
```



Eftersom referensen till objektet skickas till metoden kommer den åt objektet (utanför metoden)

- På samma sätt som för en array
- Samma problem som för en array

Gäller även strängar men dessa kan inte förändras (icke muterbara) så ingen risk.

Kedjade metodanrop

```
Complex c = new Complex(1, 2);
// Chained calls (invisible objects)
c.add(c).add(c).equals(new Complex(3, 6));


// Returns an object
Complex add(Complex other) {
   return new Complex(this.re + other.re,
                      this.img + other.img);
}
```


Om metoden returnerar en referens till ett objekt kan vi direkt anropa en metod på objektet (utan att spara referensen i en variabel).

- Sker detta i flera steg
  - Om så finns det ett antal “osynliga” mellanliggande objekt.
- Kan ge smidig kod i vissa fall.

En annan variant för att möjliggöra kedjade anrop är att returnera this

- Inga mellanliggande objekt kommer då att skapas
- Ingen bra idé för koden i bilden, där vill vi ha nya objekt (eftersom de är icke-muterbara)

Instans- och Klassmetoder

```
static int getPrecedence(String op) {
    if ("+-".contains(op)) {
        return 2;
    } else if ("*/".contains(op)) {
        return 3;
    } else if ("^".contains(op)) {
        return 4;
    } else {
        throw new RuntimeException(OP_NOT_FOUND);
    }
}
```


See Bildserie om klasser

Åtkomst för Metoder

```
public
private
protected
```


See Bildserie om klasser

Metodreferenser

```
// Use in iteration
private void renderCountry(Country c) {  // The method
     …
}

diceWars.getCountries().forEach(this::renderCountry);
```

Metodreferenser

```
// Use in JavaFX
private void animationFinished(Event e) { // The method
   from.setSelected(false);
   country.setSelected(false);
   …
}
private ParallelTransition dicesAnimation = … ;
dicesAnimation
     .setOnFinished(this::animationFinished);
```

24

Man kan ha referenser till metoder i Java

- Vi går inte in på detaljerna här (man kan spara referenser i variabler d.v.s. det finns typer för olika sorters metoder m.m.)
- En metodreferens skrivs: objektet :: metoden (dubbla kolon)
  - Objektet vi använder är this.

Användning

- För intern iteration av samlingar (forEach)
  - Man anger vilken metod som skall anropas för varje element i samlingen (elementet skickas som argument till metoden, d.v.s. metoden måste ha en parameter av elementtypen)
- I JavaFX då vi skapar händelsestyrda program

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 StackOverflowError
- … allt minne för anropsstacken är fyllt.

Rekursiva Metoder

```java
// Starter method
public int binarySearch(int[] a, int x) {
   return binarySearch(a, x, 0, a.length - 1);
}
// Need extra low and high parameters
private int binarySearch(int[] a, int x, int low, int high) {
   if (low > high) {
       return -1;
   }
   int mid = (low + high) / 2;
   if (a[mid] == x) {
       return mid;
   } else if (a[mid] < x) {
       return binarySearch(a, x, mid + 1, high);  // Recursive call
   }else { // last possibility: a[mid] > x
       return binarySearch(a, x, low, mid - 1);   // Recursive call
   }}
```

26

En rekursiv metod är en metod som anropar sig själv

- För att inte få StackOverflow måste argumenten förändras vid varje anrop.

Rekursiva metoder kan ibland ge eleganta lösningar till knepiga problem

- Ofta då man har mer komplicerade datastrukturer
- I Bilden: Rekursiv variant av binärsökning
  - Lite tydligare lösning än iterativa ....
  - … tyvärr ineffektiv. Många metodanrop!