
Slides adapted from Algorithms 4th Edition, Sedgewick.

# ID1020 Algoritmer och Datastrukturer

Dr. Jim Dowling (Examinator)
jdowling@kth.se

Dr. Johan Karlandar (Examinator)
johank@nada.kth.se

Dr. Per Brand (Teacher)
pbrand@kth.se

# Kursens överblick

- Programmering och lösa problem med applikationer
- Algoritm: en metod för att lösa ett problem
- Datastruktur: en struktur där man lagrar information
- Utvärdera algoritmer: främst beräkningskomplexitet

| Område | kapitel i boken | Algoritmer och Datastrukturer |
|---|---|---|
| datatyper | 1 | stack, queue, bag, union-find |
| sortering | 2 | quicksort, mergesort, heapsort, prioritetkö |
| sökning | 3 | BST, red-black BST, hashtabell |
| grafer | 4 | BFS, DFS, Prim, Kruskal, Dijkstra |

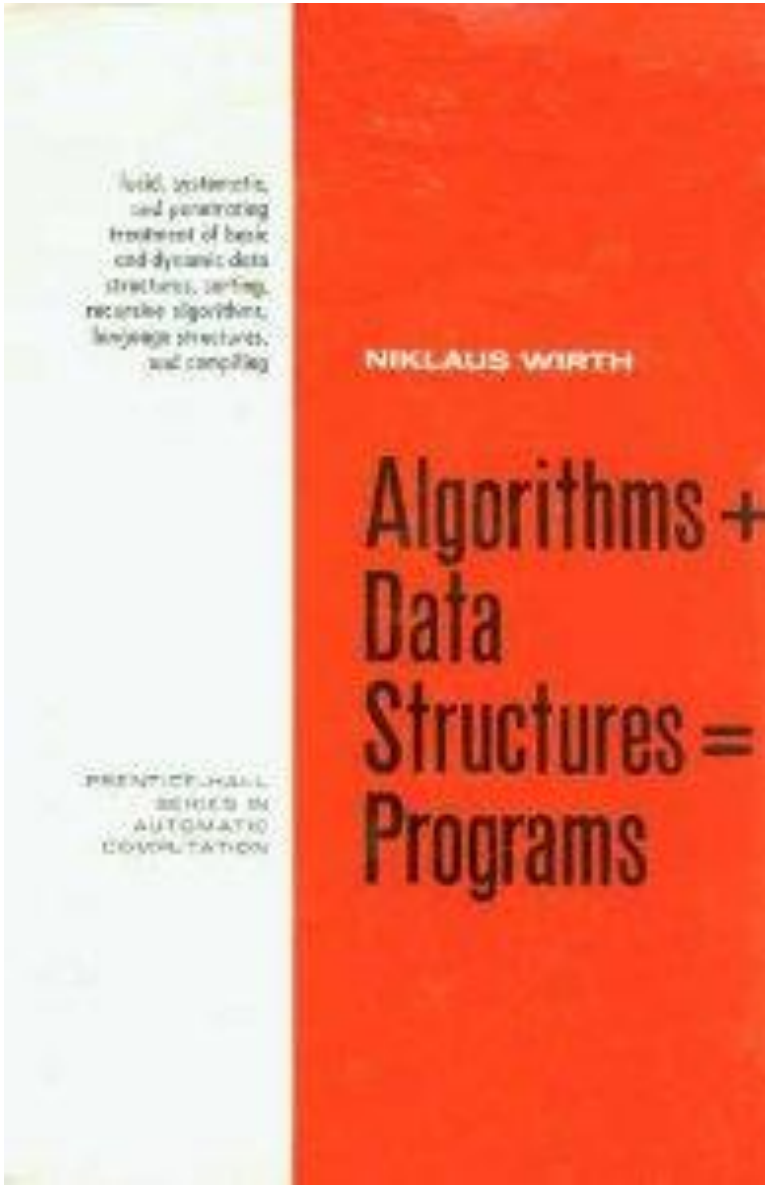
# Varför läsa algoritmer och datastrukturer?

- Stor påverkan på omvärlden.

  - Internet. Web sökning, datornätverk, browsers, ...
  - Datorer. Hårdvara, grafik, datorspel, lagring, ...
  - Security. Cell phones, e-commerce, voting machines, ...
  - Biologi. Helgenomsekvensering, proteinveck,..
  - Sociala nätverk. Recommendationer, newsfeeds, reklam, ...
  - Fysik. Large Halydron Collider – Higgs Boson, N-body simulering, particle collision simulering, ...

# För att kunna bygga programvara

- Algoritmer + datastrukturer = progamvara
  - Niklaus Wirth, 1976



# För att bli en bättre ukvecklare

- "I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

— Linus Torvalds (skapare av Linux)

# Datadriven vetenskap

- Beräkningsmodeller håller på att ersätta anlaytiska modeller inom naturvetenskap

$$\frac{dx}{dt} = x(\alpha - \beta y)$$

$$\frac{dy}{dt} = -y(\gamma - \delta x)$$

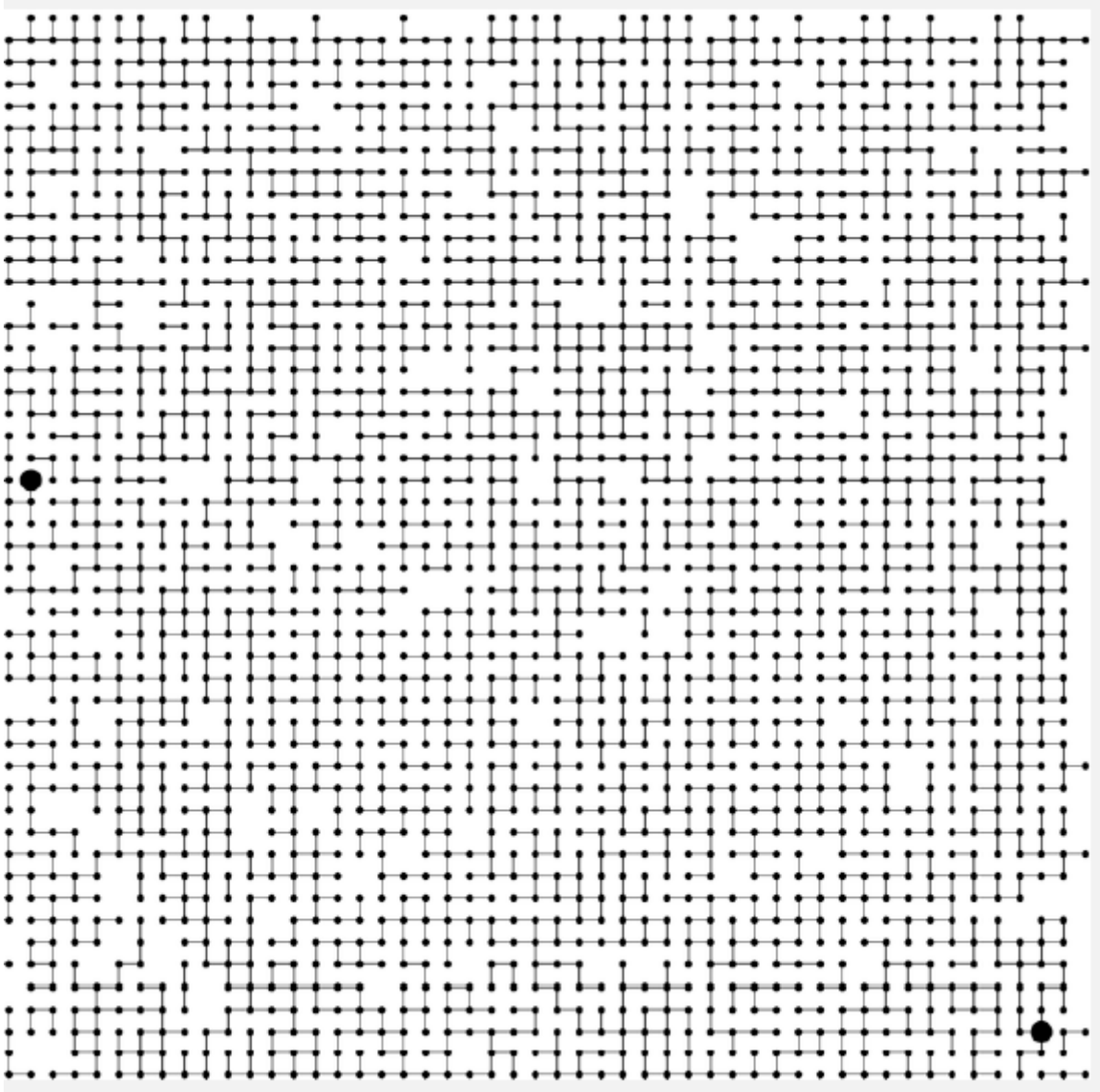
1900-talet (ekvationsbaserad)

```
procedure ACO_MetaHeuristic
  while(not_termination)
    generateSolutions()
    daemonActions()
    pheromoneUpdate()
  end while
end procedure
```

2000-talet (algoritmsbaserad)

# Lösa svåra problem..

T.ex.,
nätverkskonnektivitet



# Varför läsa algoritmer?

- För att det lönar sig!


# Förkunskap

- En avklarad kurs i programmering (t.ex. ID1018)
  - loopar, arrayer, funktioner, objektorienterad programmering, strängar
- Erfarenhet med programmering i Java

# Resurser

- Canvas
  - Alla uppgifter ska lämnas in på Canvas
- Inget grupparbete
  - Kursen examineras individuellt
- Hemsidan
  - www.it.kth.se/courses/ID1020

# Mer kurs info

- Krav
  - aktuell laborationskurs
  - tentamen

- Labbassistenter
  - Kamal Hakimzadeh
  - **Mahmoud Ismail**
  - Robin Andersson
  - Alex Ormenisan

- **OBLIGATORISK TENTAANMÄLAN**

# Kursboken

- SKAFFA OCH LÄS BOKEN!
  - **Algorithms 4th Edition, Sedgewick.**
- Tips
  - Börja med labbarna och projekten i tid!
  - Gör hemuppgifterna på Bilda – de är inte särskilt svåra

# Examination

- Hemupggifter (10%)
  - 1:e och 8:e September. Lämnas in på Canvas.
- 5 laborationer (50%)
  - 6/7:e, 14:e (Canvas), 20/21:e September
  - 26/28:e September, 9/13:e Oktober
- 2 programmeringsprojekt (40%)
  - 4/5:e Oktober
  - 6/9:e November
- Tentamen (3,0 HP)

Hemuppgift (4,5 HP)

# Programeringsmodeller och Data Abstraktion

- Rekursion (kap 1.1, sida 25 i boken)
  - En metod kan anroper sig själv!

- Läs kap. 1.1 och 1.2 från
  Algorithms 4th Edition, Sedgewick och Wayne.

# Grundläggande Java kunskap

| term | examples | definition |
|---|---|---|
| *primitive data type* | `int double boolean char` | a set of values and a set of operations on those values (built in to the Java language) |
| *identifier* | `a abc Ab$ a_b ab123 lo hi` | a sequence of letters, digits, _, and $, the first of which is not a digit |
| *variable* | [*any identifier*] | names a data-type value |
| *operator* | `+ - * /` | names a data-type operation |
| *literal* | `int` `1 0 -42`<br>`double` `2.0 1.0e-15 3.14`<br>`boolean` `true false`<br>`char` `'a' '+' '9' '\n'` | source-code representation of a value |
| *expression* | `int` `lo + (hi - lo)/2`<br>`double` `1.0e-15 * t`<br>`boolean` `lo <= hi` | a literal, a variable, or a sequence of operations on literals and/or variables that produces a value |

# Grundläggande Java kunskap

| statement | examples | definition |
|---|---|---|
| *declaration* | `int i;`<br>`double c;` | create a variable of a specified type, named with a given identifier |
| *assignment* | `a = b + 3;`<br>`discriminant = b*b - 4.0*c;` | assign a data-type value to a variable |
| *initializing declaration* | `int i = 1;`<br>`double c = 3.141592625;` | declaration that also assigns an initial value |
| *implicit assignment* | `i++;`<br>`i += 1;` | `i = i + 1;` |
| *conditional* (`if`) | `if (x < 0) x = -x;` | execute a statement, depending on boolean expression |
| *conditional* (`if-else`) | `if (x > y) max = x;`<br>`else max = y;` | execute one or the other statement, depending on boolean expression |

# Arrayer

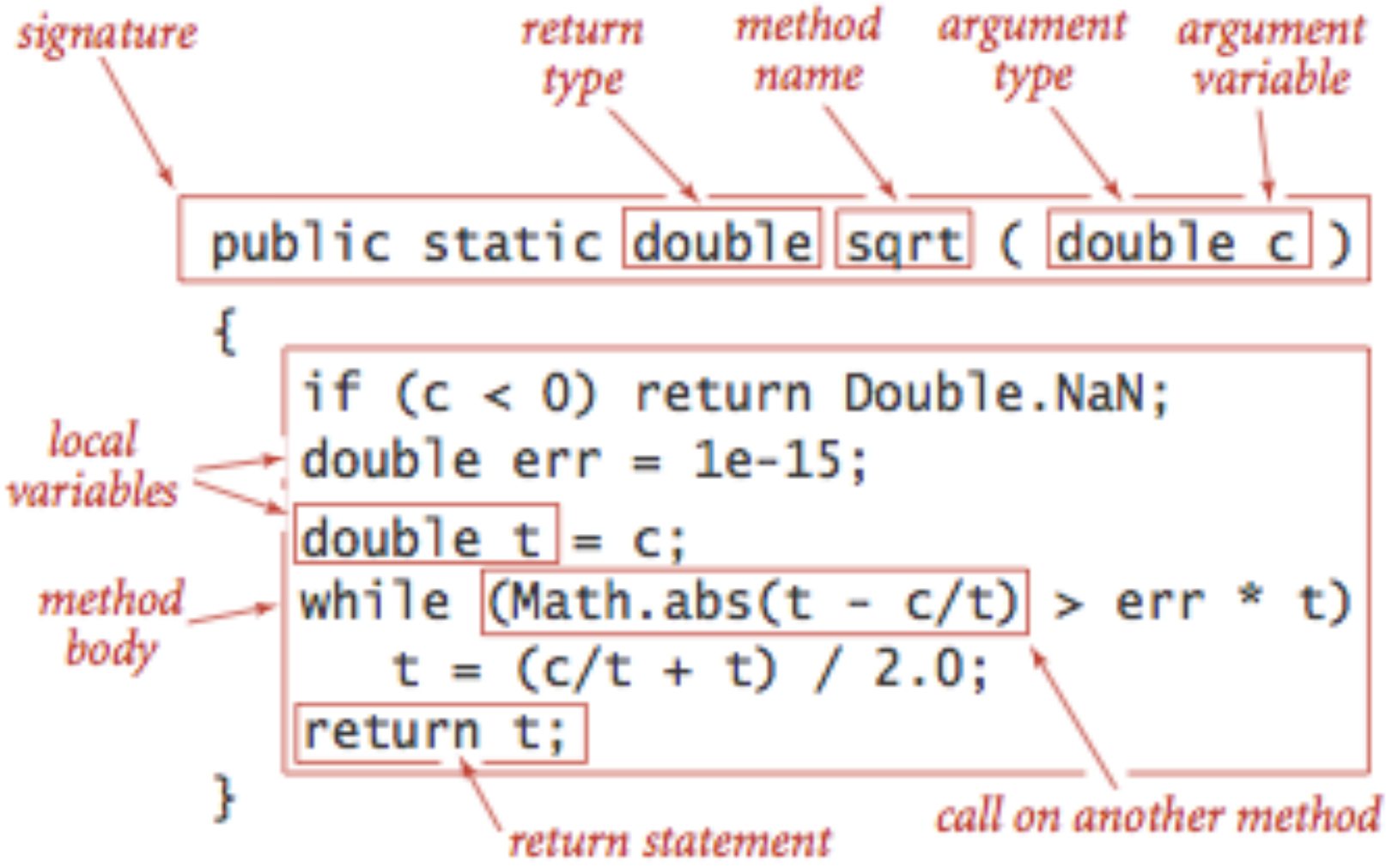
```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

```
double[] a = new double[N];
```

# Aliasing

```
int[] a = new int[N];
  ...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678; // vad har a[i] för värde nu?
```

# Statiska Metoder



# Egenskaper av metoder

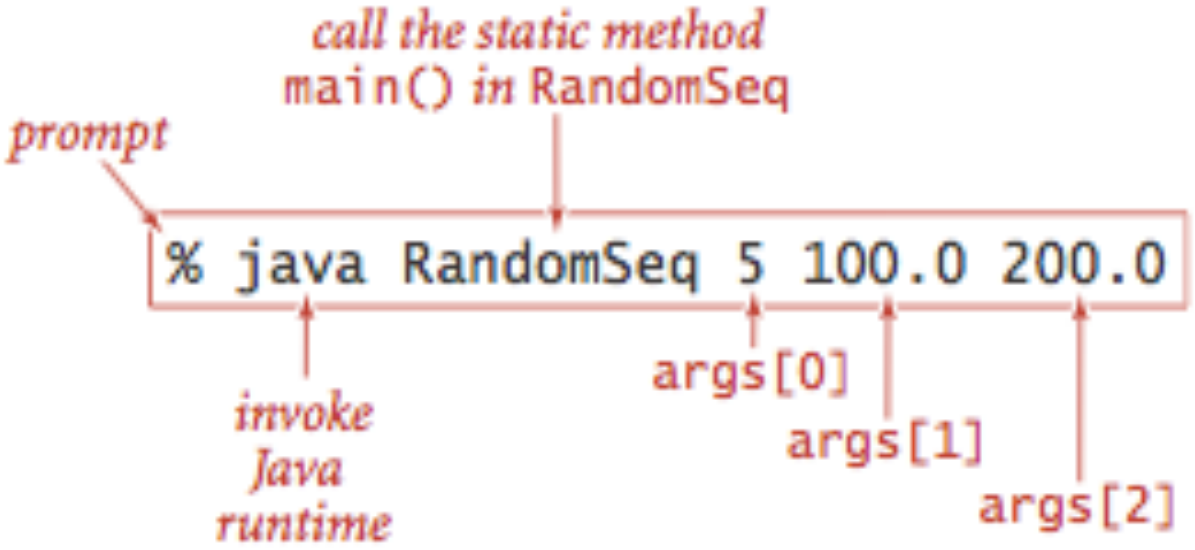
- Argument
  - *pass-by-value* (primitivtyper)
  - *pass-by-reference* (objekt)
- Metodnamn kan vara överlagrad (*overloaded*)
  - *T.ex. Math.min(int x, int y), Math.min(double x, double y)*
- Metoder har bara ett returvärde, men kan returnera från många olika ställen i en metod.
- En metod kan ha *bi-effekter*
  - T.ex., uppdatera attribut i objekt

# Input och Output i Java

- Ett Java program kan tar input värden från:
  1. *kommando argument*
     - public void static main(String[] args)
  2. *miljövariabler*
     - java –Djava.library.path=/home/jim/libs –jar MyProgram.jar
  3. *standard-input stream* (stdin)
     - en abstrakt ström av karaktärer

- Ett Java program kan skriva output värden till:
  1. *standard-output stream* (stdout)

# Exekvera ett Java program



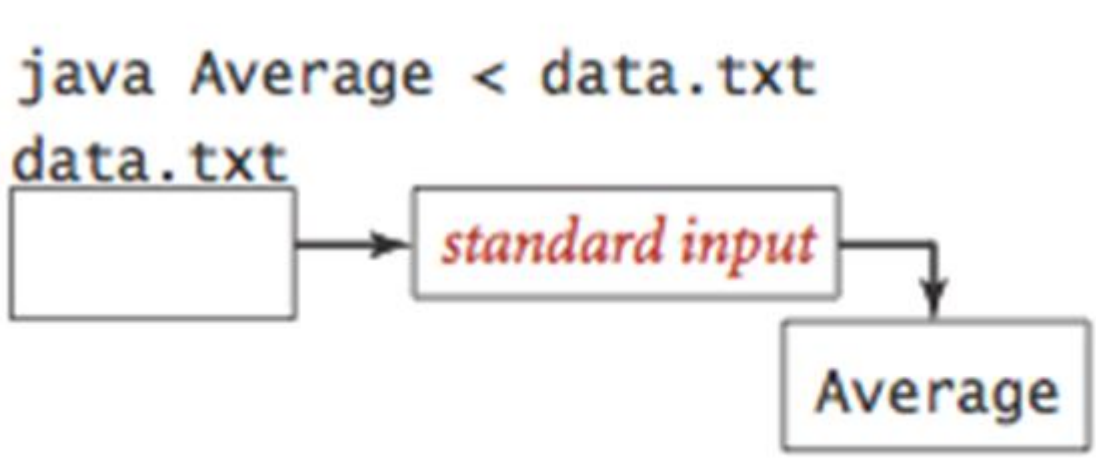
# Formaterade Output

| type | code | typical literal | sample format strings | converted string values for output |
|---|---|---|---|---|
| int | d | 512 | "%14d" | "  512" |
| | | | "%-14d" | "512  " |
| double | f | 1595.1680010754388 | "%14.2f" | "  1595.17" |
| | | | "%.7f" | "1595.1680011" |
| | e | | "%14.4e" | "  1.5952e+03" |
| String | s | "Hello, World" | "%14s" | "  Hello, World" |
| | | | "%-14s" | "Hello, World  " |
| | | | "%-14.5s" | "Hello  " |

# Omdirigering från stdinput

redirecting from a file to standard input

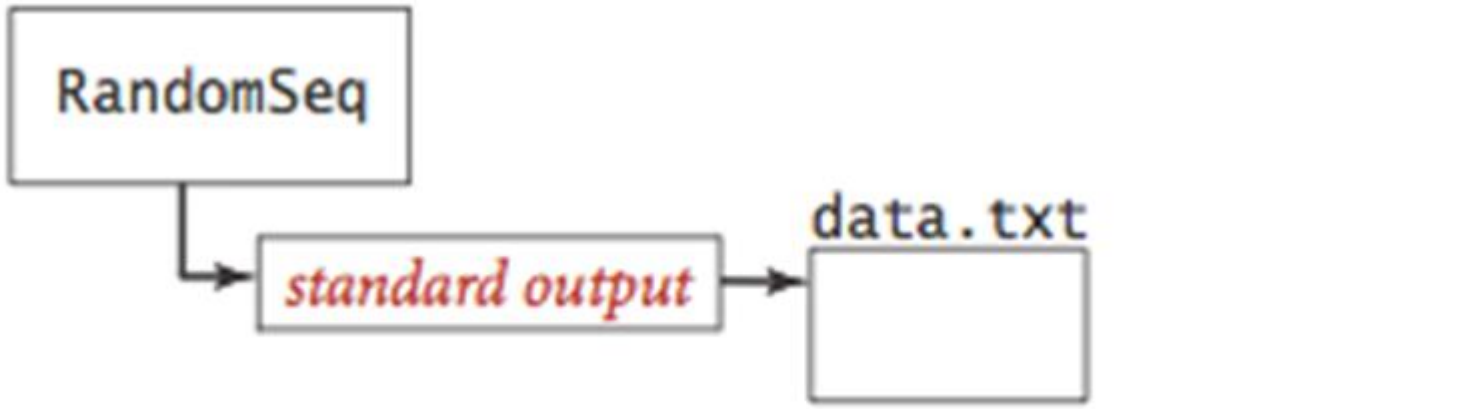
```
% java Average < data.txt
```



# Omdirigering till stdout

redirecting standard output to a file

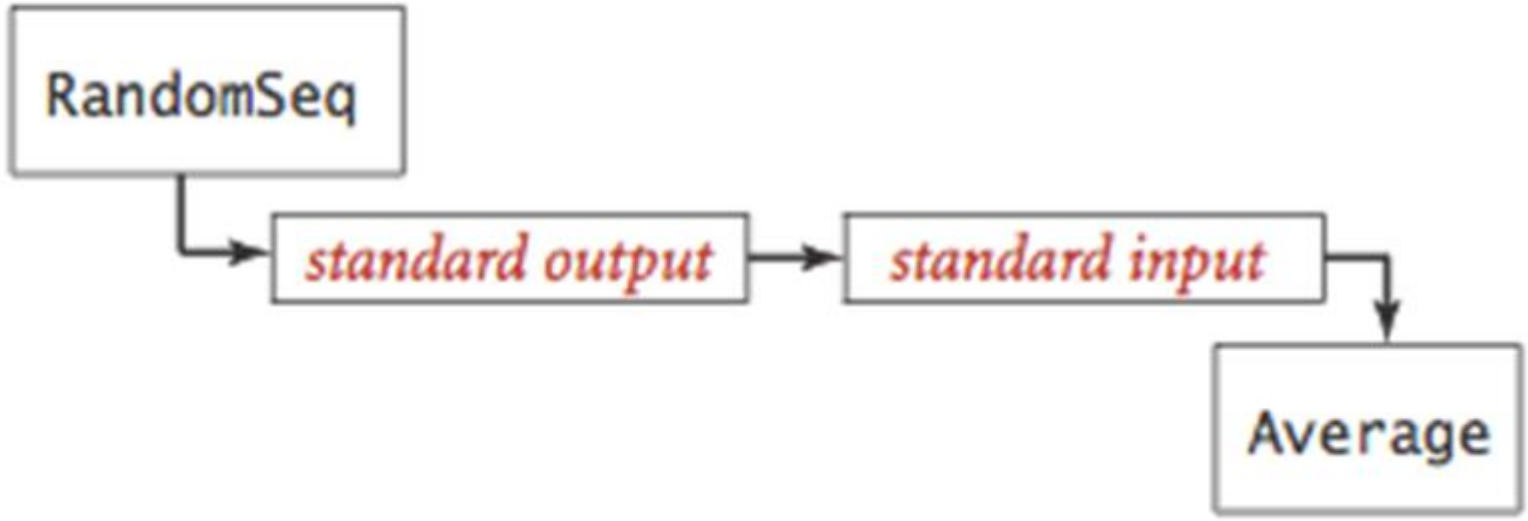
```
% java RandomSeq 1000 100.0 200.0 > data.txt
```



# Piping output från ett program till input av ett annat program

piping the output of one program to the input of another

```
% java RandomSeq 1000 100.0 200.0 | java Average
```



# API:er och Objektorienterad Programmering

# Data abstraktion

- Objektorientierad design
  - Abstrakt data typer

- Ett *Application Programming Interface (***API***)* är ett gränssnitt som specificierar beteendet av en abstract datatyp (ett kontrakt).

- Ett API inkapsulerar (*encapsulates*) beteendet av en abstrakt datatyp.
  - Klienten vet inget om hur abstrakt datatypen ser ut inuti.

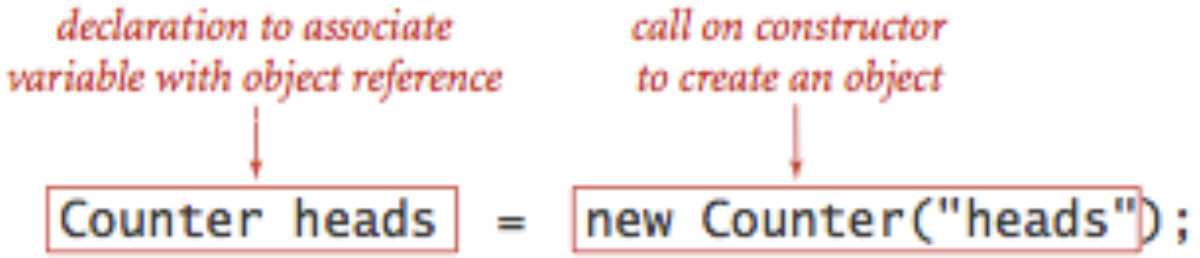
# Klass och Objekt

- Vilka operationer finns i API:et till Counter klassen?

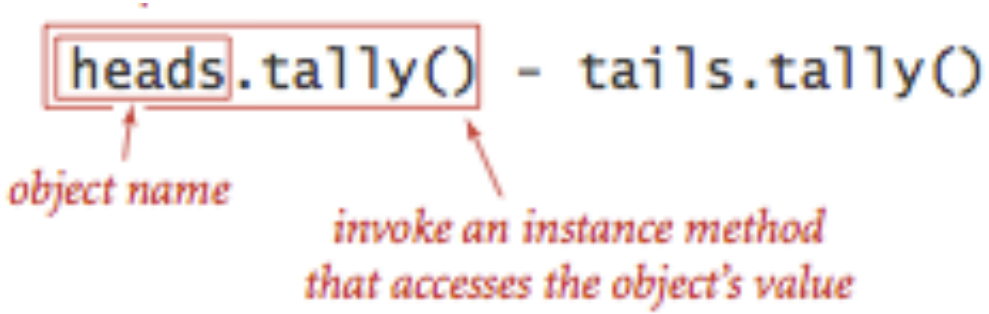
| public class Counter | | |
|---|---|---|
| | Counter(String id) | *create a counter named* id |
| void | increment() | *increment the counter by one* |
| int | tally() | *number of increments since creation* |
| String | toString() | *string representation* |

# Counter Klasss API:n

- Skapa objekt




- Anropa metod



# Hur skriver man ett bra API?

- Inkapsulering
- Tydligt kontrakt
- Ger klienten vad den behöver, inte mer.
- Dåliga egenskaper av ett API
  - Repetiton av metoder DRY = do not repeat yourself
  - För svårt att implementera
  - För svårt att använda för klienten
  - För smal (*narrow*)– saknar metoder som klienten behöver
  - För bred (*wide*) – inkluderar metoder som klienten inte behöver
  - För allmän (*too general*) – inga nyttiga abstraktioner
  - För specificik – abstraktionerna hjälper för få klienter
  - För kopplad till en representation – klienter måste veta detaljer om en representation

# API Design – String Klass

| public class String | |
|---|---|
| String() | *create an empty string* |
| int length() | *length of the string* |
| int charAt(int i) | *i th character* |
| int indexOf(String p) | *first occurrence of* p *(-1 if none)* |
| int indexOf(String p, int i) | *first occurrence of* p *after* i *(-1 if none)* |
| String concat(String t) | *this string with* t *appended* |
| String substring(int i, int j) | *substring of this string (*i*th to* j*-1st chars)* |
| String[] split(String delim) | *strings between occurrences of* delim |
| int compareTo(String t) | *string comparison* |
| boolean equals(String t) | *is this string's value the same as* t*'s ?* |
| int hashCode() | *hash code* |

**Java String API (partial list of methods)**

Behöver klienter metoden? `int indexOf(String p)`

# Rekursion

# Ett enkelt rekursivt program

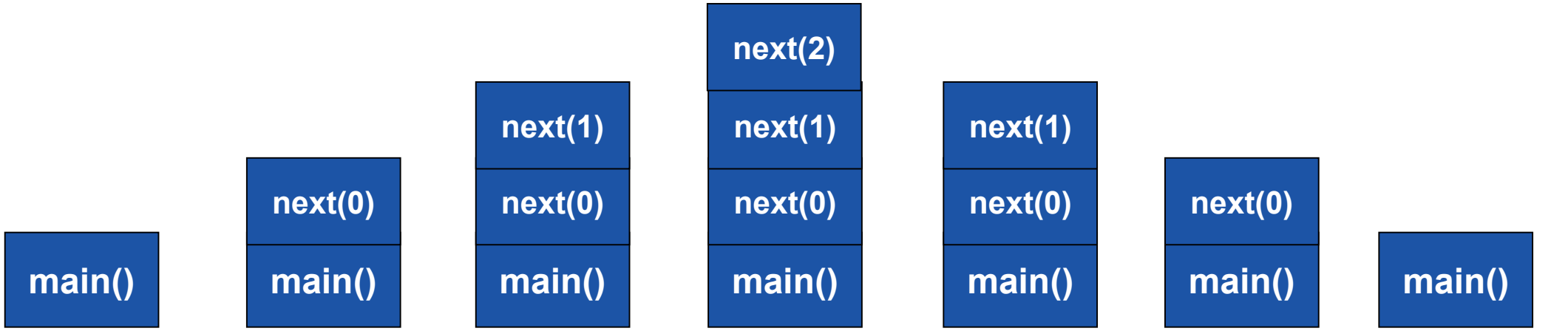
```
public class Recursion
{
  public static void main (String args[])
  {
        next(0);
  }
  public static void next(int index)
  {
        StdOut.print(index);
        if (index < 2) {
            next(index+1);
        } else {
            StdOut.println(" klar");
        }
  }
}
```

rekursion här (ett rekursivt anrop)

"basfall" (base case) här

Programmet skriver ut:
012 klar

# Visualisera rekursion med tid som en ”Stack”




**Tid 0:**
**Push: main()**

**Tid 1:**
**Push: next(0)**

**Tid 2:**
**Push: next(1)**

**Tid 3:**
**Push: next(2)**

**Tid 4:**
**Pop next(2)**

**Tid 5:**
**Pop next(1)**

**Tid 6:**
**Pop next(0)**

Basfallet: villkorn blir ”false”, else satsen exekveraras och next(2) returneras.

# Rekursion

- Överväga följande serien:

  1, 3, 6, 10, 15….

- Skriv ett program som beräknar nummret N i serien:
  1. for-loop
  2. while-loop
  3. rekursion

# Hitta n:te termen (for-loop stigande)

```
int triangel(int n) {
    int summa= 0;
    for (int i=0; i<=n; i++) {
        summa = summa + i;
    }
    return summa;
}
```

# Hitta n:te termen (while-loop nedstigande)

```
int triangel(int n) {
    int total = 0;
    while (n > 0){
        total = total + n;
        --n;
    }
    return total;
}
```

# Nu, samma program med rekursion

# Hitta n:te termen (rekursion)

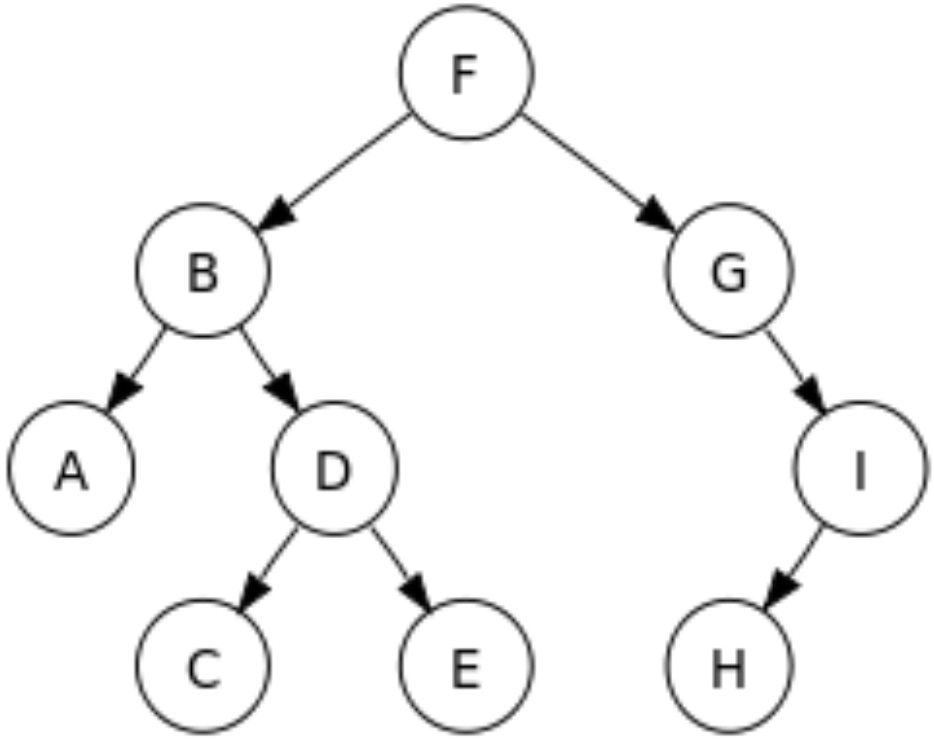
```
int triangel(int n) {
     if (n == 1){
       return 1;
     } else {
        return (n + triangel(n-1));
     }
}
```

basfall

rekursion

# Rekursion - motivation

- I datorvetenskap, visa problem är lättre att lösa med hjälp av rekursiva funktioner:
  - Traversera ett filsystem.
  - Traversera ett träd av sökresultat.



# Fakultet

- En Fakultet defineras som:

  n!  = n * (n-1) * (n-2) …. * 1;

- T.ex.:

  1! = 1 (basfall)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

- Vi ska försöka lösa problemet i två delar:
  1. Vad programmet kan lösa i en sats (ett basfall)
  2. Vad man kan lösa med många likadana satsar
     - Sedan anropa en kopia av funktionen själv för att göra nästa “steg”

# Fakultet Program

```
public static int computeFactorialWithLoop(int n)
{
     int factorial = n;
     for (int i = n - 1; i >= 1; i--) {
          factorial = factorial * i;
     }
     return factorial;
}
```

```
public static int findFactorialRecursion(int n)
{
    if ( n == 1 || n == 0) {
          return 1;
    } else {
          return (n * findFactorialRecursion(n-1));
    }
}
```

# Fibonaccis talföljd

- Fibonaccis talföljd
  - Varje tal är summan av de två föregående Fibonaccitalen
    - t.ex., 0, 1, 1, 2, 3, 5, 8, 13, 21…

      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci( n – 2 )

    - fibonacci(0) och fibonacci(1) är basfallen

$$F(n) = \begin{cases} 0 & \text{om } n = 0; \\ 1 & \text{om } n = 1; \\ F(n-1) + F(n-2) & \text{om } n > 1. \end{cases}$$

# Rekursion och Loopar

- Rekursion
  - Byggar på villkorssatser (`if`, `if…else` or `switch`)
  - Repetition med hjälp av upprepade metodanropar
  - Terminerar när basfallet är nått/sant
  - Kontrollerar repetition genom att partitionerar ett problem i flera enklare problem
- Loopar
  - Byggar på `for`, `while` or `do…while`
  - Repetition med hjälp av explicit representation av en repetitionskodblock
  - Terminerar när loop-villkor blir falsk eller “break” anropas.
  - Kontrollerar repetition med hjälp av en räknare

# Rekursion vs. Loopar (fort.)

- Rekursion
  - Mer overhead än iteration
    - ett undantag är med svansrekursion i vissa implementeringar
  - Kräver mer stackminne
    - ett undantag är med svansrekursion i vissa implementeringar
  - Går att lösa med loopar
  - Kan oftast skrivas med ett få antal rader källkod

# Svansrekursion (*Tail recursion*)

- En optimerad sorts rekursion då sista operationen i en funktion är ett rekursivt anrop.
  - Anropet är det sista som funktionen gör, så det kan ersättas med ett hopp, då ingen returadress behöver sparas på stacken. Detta kallas för svansanropsoptimering.

  ⇒ stacken växer inte längre proportionellt mot antal rekursiv anrop

```
int triangel(int n) {
    if (n == 1){
      return 1;
    } else {
       return (n + triangel(n-1));
    }
// Inga mer operationer i funktionen
}
```

sista operation är ett rekursivt anrop

Om man lägger till en/fler sats efter recursivt anropoet är det inte längre svansrekursion

# Rekursion sammanfattning

- Rekursiv tanke: reducerar problemet till ett enklare problem med samma struktur

- Basfall: det måste finnas ett fall som inte leder till rekursivt anrop