Slides adapted from Algorithms 4th Edition, Sedgewick.

ID1020 Algoritmer och Datastrukturer

Robert Rönngren(Examinator)
rron@kth.se

Overview

- Algorithms: methods to solve a problem
- Data structure: a structure to store information that can be used to implement/support an algorithm
- Abstract Data Types (ADTs): a datatype with methods implementing an algorithm, well-defined API
- You should learn to: Understand, Select, Implement, Modify

Area	Chapter	Algorithms och Data structures
Basic data types	1	stack, queue, bag, union-find
Sorting	2	quicksort, mergesort, heapsort, priority queues
Searching	3	BST, red-black BST, hash tables
Graphs	4	BFS, DFS, Prim, Kruskal, Dijkstra

Definition of the concept of algorithm

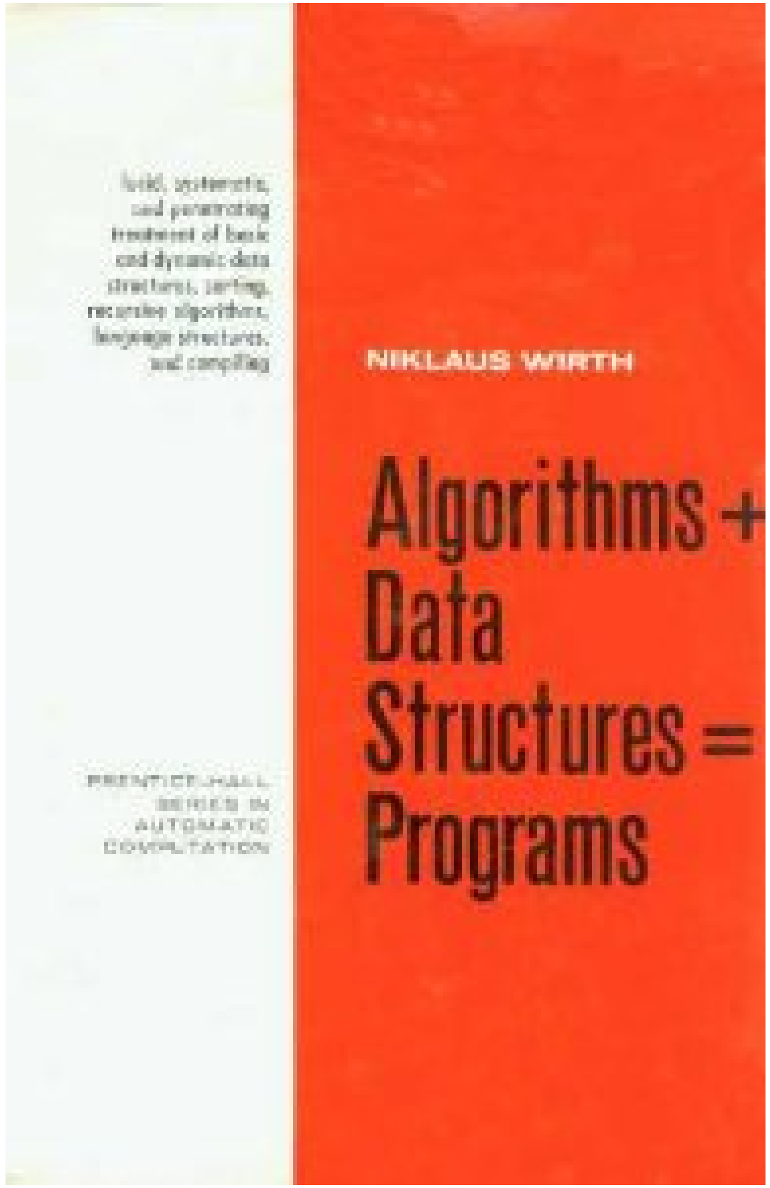
- An algorithm in mathematics and computer science is a (limited) set of well-defined instructions to solve a problem, which from given starting states (i.e. given set of input) with certainty leads to specific end states (i.e. output)
- A problem with and a specific input is called an instance of the problem
- An algorithm is **correct** if the algorithmic procedure **stops** for each problem instance and returns the **required output**. The algorithm **solves** the problem.
- A programmer's job is to design and implement algorithms that are **correct**, **simple** and **efficient**.

Why should you learn this?

- Basis for all types of applications
  - Internet: Web search, computer networks, browsers, ...
  - Computers: hardware, grafics, games, storage, ...
  - Security: Cell phones, e-commerce, voting machines, ...
  - Biology: genome sequencing, large data sets,..
  - Sociala networks: Recommendations, newsfeeds, commercials, ads, ...
  - Physics: Large Halydron Collider – Higgs Boson, N-body simulation, particle collision simulation, ...

To construct software

- Algorithms + data structures = software
  - Niklaus Wirth, 1976



To become a better programmer

- “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 (creator of Linux)

Data driven science

- Computational models are replacing analythical models

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

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

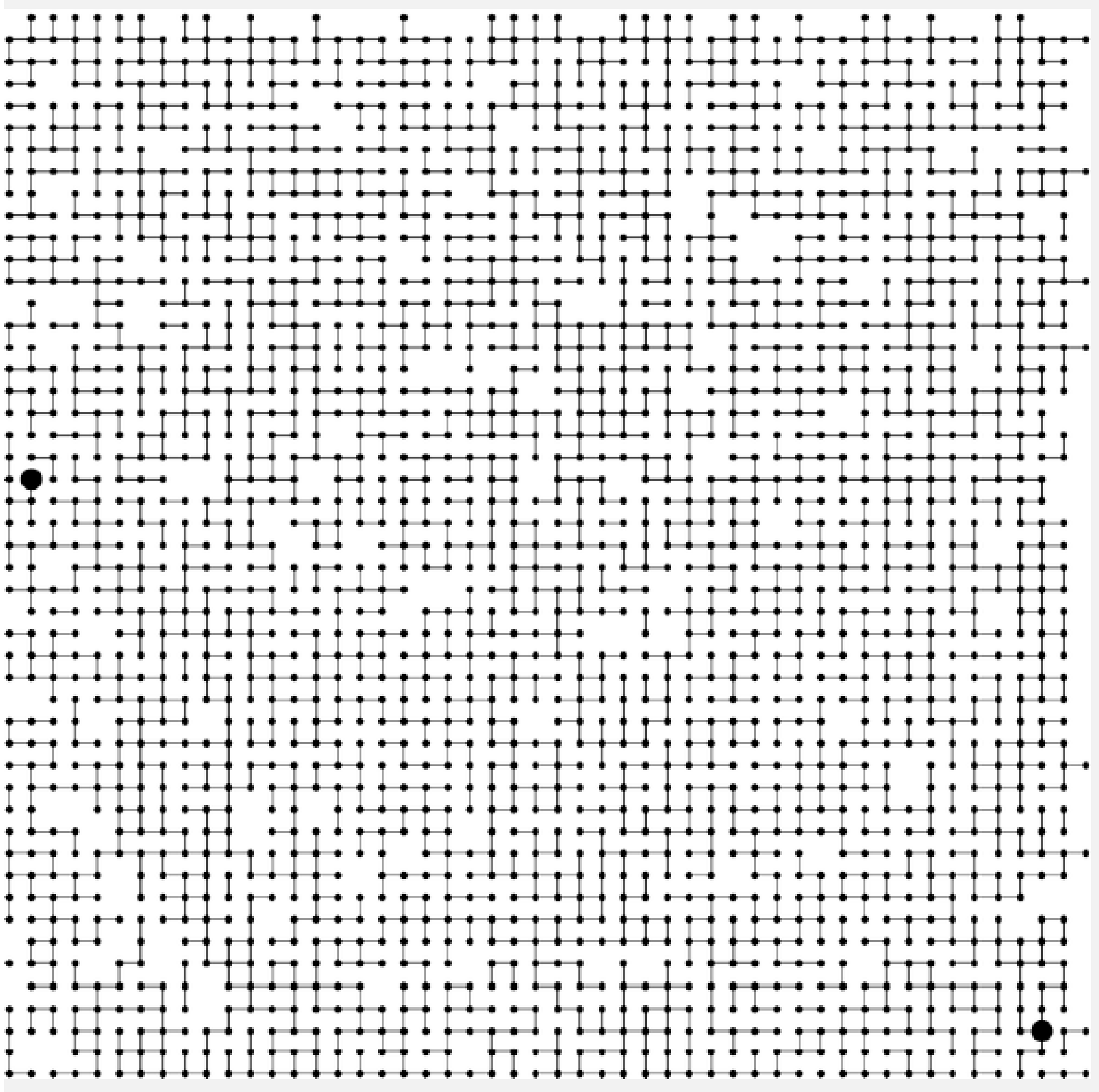
1400-2000

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

2000-

Solve hard problems..

Eg. network connectivity



Why study algoritms?

- It pays off!

Pre-requisites

- A finished course in the basics of programming (ID1018)
  - loops, arrays, functions/methods, object oriented programming, strings, (files)
- Knowledge of Java (and C)

Resources

- Canvas
  - All material: lecture notes, quizes, labs in Canvas

- No teamwork
  - Individual examination

More info

- Requirements
  - Quizes (to access material)
  - Labs – ”kryssfrågor” 80%
  - Written exam
  - Labs to be completed in the fall semester

- TAs
  - Fredrik Lundevall
  - Kamal Hakimzadeh
  - Mahmoud Ismail

- **Remember to sign up for the exam**

Book

- Get access to the book (buy it)!
  - ***Algorithms 4th Edition, Sedgewick.***

- Start reading, do quizes, work with labs in time

Note on API for the examples in the book

- The book uses a library which implements parts of the C-library for among other things input/output!
- Check the web resource pages – ask at the guidance sessions

Programmering model and Data Abstraction

- Recursion (ch 1.1, page 25)
  - A method/function can call itself!

- Read ch. 1.1 and 1.2
  Algorithms 4th Edition, Sedgewick och Wayne.

Basic JAVA (1)

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`
	`double`
	`boolean`
	`char`
		`1 0 -42`
		`2.0 1.0e-15 3.14`
		`true false`
		`'a' '+' '9' '\n'`
			source-code representation of a value

expression	`int`
	`double`
	`boolean`
		`lo + (hi - lo)/2`
		`1.0e-15 * t`
		`lo <= hi`
			a literal, a variable, or a sequence of operations on literals and/or variables that produces a value

Basic Java (2)

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

Arrays

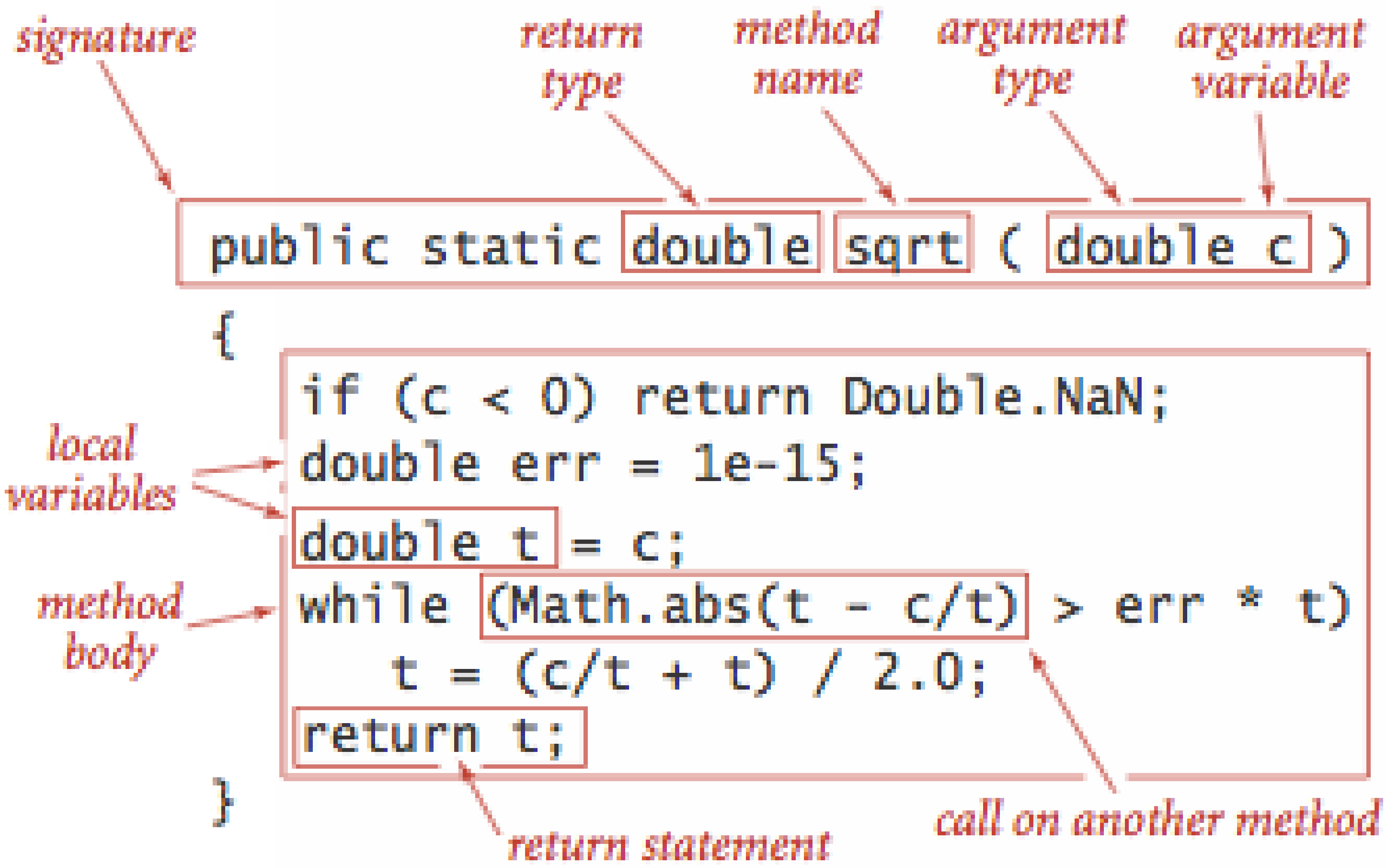
```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

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

Aliasing

```
int[] a = new int[N];
 ...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678; // What’s the value of a[i] after this
assignment?
```

Static Methods



Methods

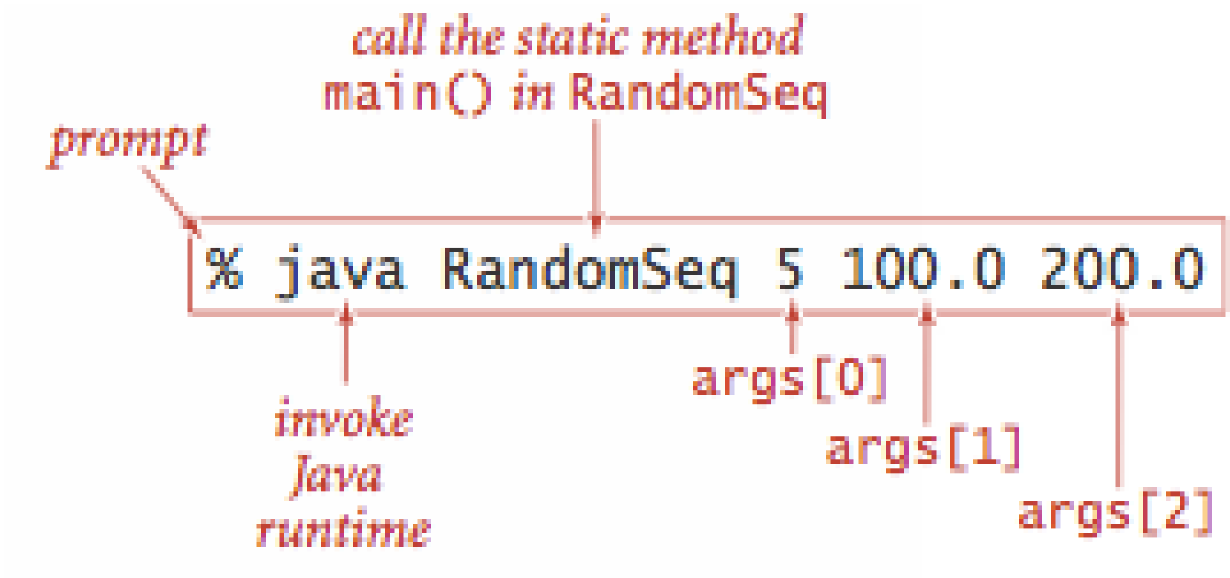
- Arguments
  - *pass-by-value* (primitive type)
  - *pass-by-reference* (object, array)
- Method names may be *overloaded*
  - *Eg. Math.min(int x, int y), Math.min(double x, double y)*
- Methods can only return one thing, but they may have many return statements
- A method may have side-effects
  - Eg., uppdate attributes of an object, order elements in an array

Input and Output in Java (processes)

- Without extra code a Java program may access input via:
  1. *command line arguments*
     - public void static main(String[] args)
  2. *Environment variables*
     - java –Djava.library.path=/home/jim/libs –jar MyProgram.jar
  3. *standard-input stream* (stdin)
     - An abstract stream of characters

- A Java program can write output to:
  1. *standard-output stream* (stdout)
  2. (stderr)

Execute a Java program



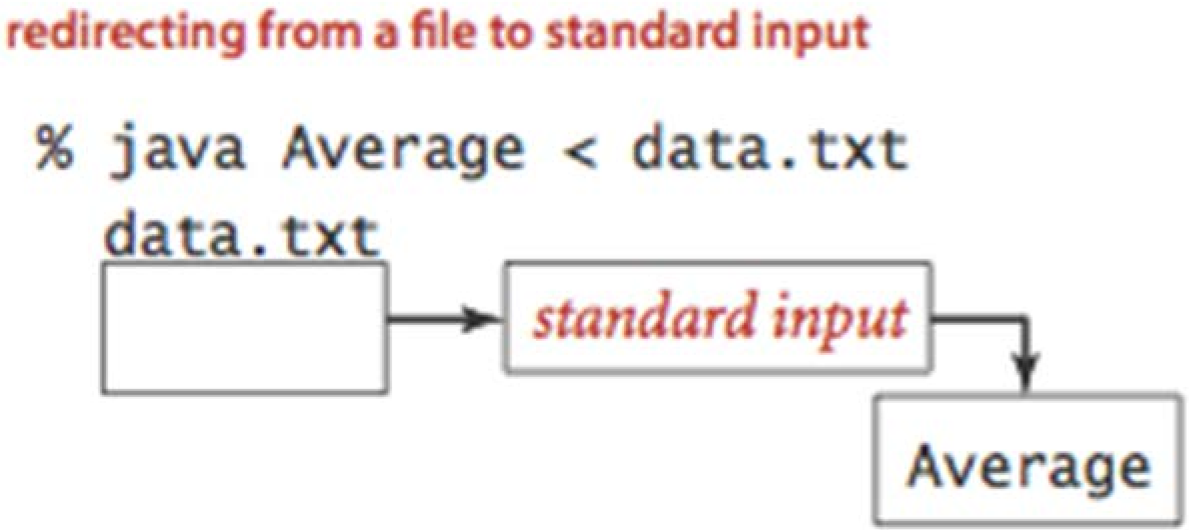
Formated Output

type	code	typical literal	sample format strings	converted string values for output
`int`	`d`	`512`	`"%14d"` `"%-14d"`	`"           512"` `"512           "`

`double`	`f`
	`e`
		`1595.1680010754388`	`"%14.2f"`
			`"%.7f"`
			`"%14.4e"`
				`"       1595.17"`
				`"1595.1680011"`
				`"    1.5952e+03"`

`String`	`s`	`"Hello, World"`	`"%14s"`
			`"%-14s"`
			`"%-14.5s"`
				`"  Hello, World"`
				`"Hello, World  "`
				`"Hello         "`

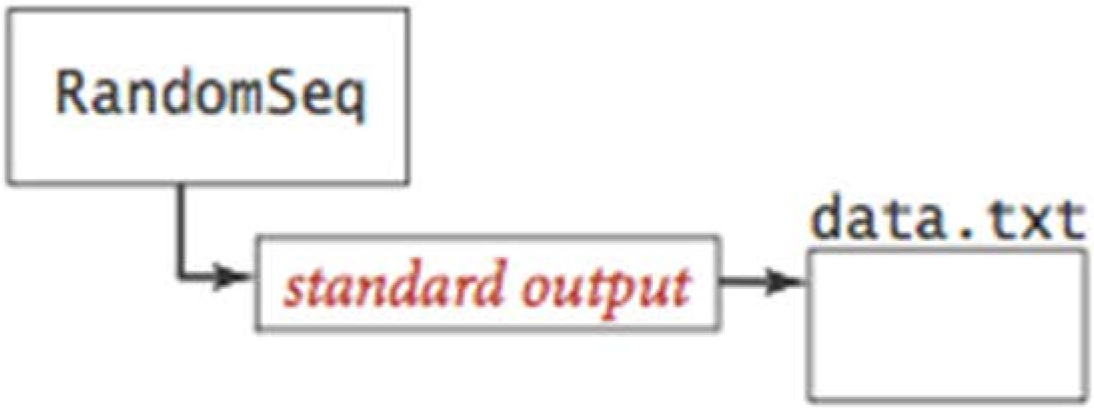
Redirect from stdin



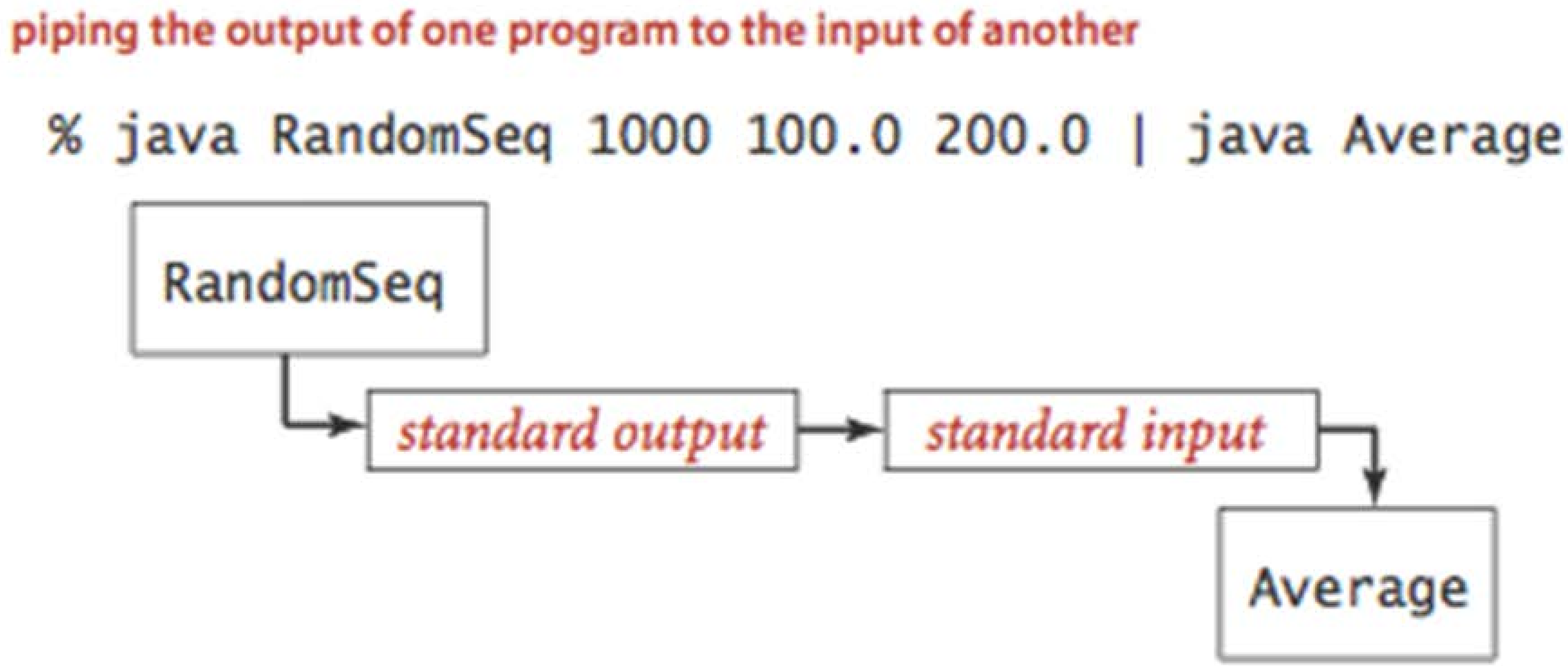
Redirect to stdout

redirecting standard output to a file

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



Piping output from a program to the input of another program



API:s and Object oriented Programming

Data abstraction

- Object oriented design
  - Abstract data types

- An *Application Programming Interface (***API***)* is an interface defining the behaviour of an abstract data type (ADT) (contract).

- An API *encapsulates* the behavious/implementation of an abstract data type (ADT).
  - The client does not (need not) know anything of the internal implementation of the ADT.

Classes and Objects

- Class
  - Template for objects
  - Class methods (only one instance in the class)
  - Class variables (only one instance in the class)
- Instance/Object
  - Objects are instantiated from the class
  - Can access class methods and variables
  - Each instance has its own set of instance methods and instance variables

Class and Object

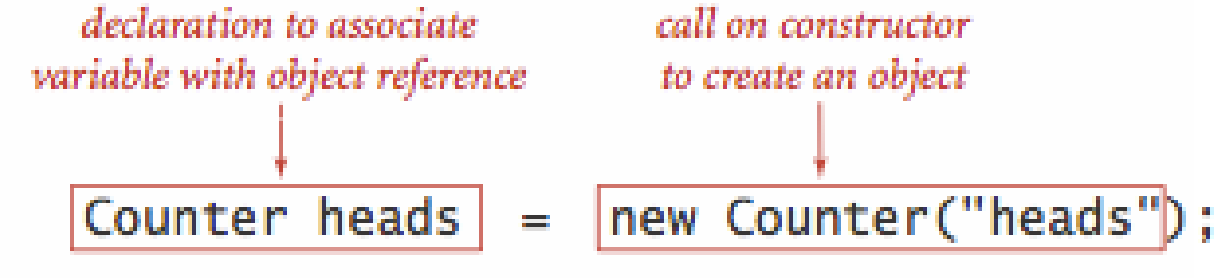
- Operations in the API of the Counter class?

`public class Counter`

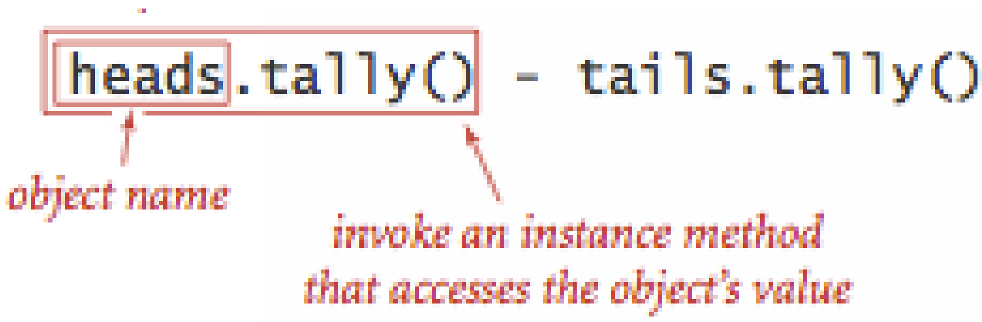
	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

Counter class API

- Create/instantiate an object



- Invoke method



How to design good APIs?

- Encapsulation
- Clear contract
- Give the client what is needed – no more.
- Bad APIs
  - Duplication of methods
  - Too hard to implement, making it difficult or impossible to develop.
  - Too hard to use, leading to complicated client code.
  - Too narrow, omitting methods that clients need.
  - Too wide, including a large number of methods not needed by any client.
  - Too general, providing no useful abstractions.
  - Too specific, providing an abstraction so diffuse as to be useless.
  - Too dependent on a particular representation, therefore not freeing client code from the details of the representation.

API Design – String Class

`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-1`*st 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)

Does the client need ? `int indexOf(String p)`

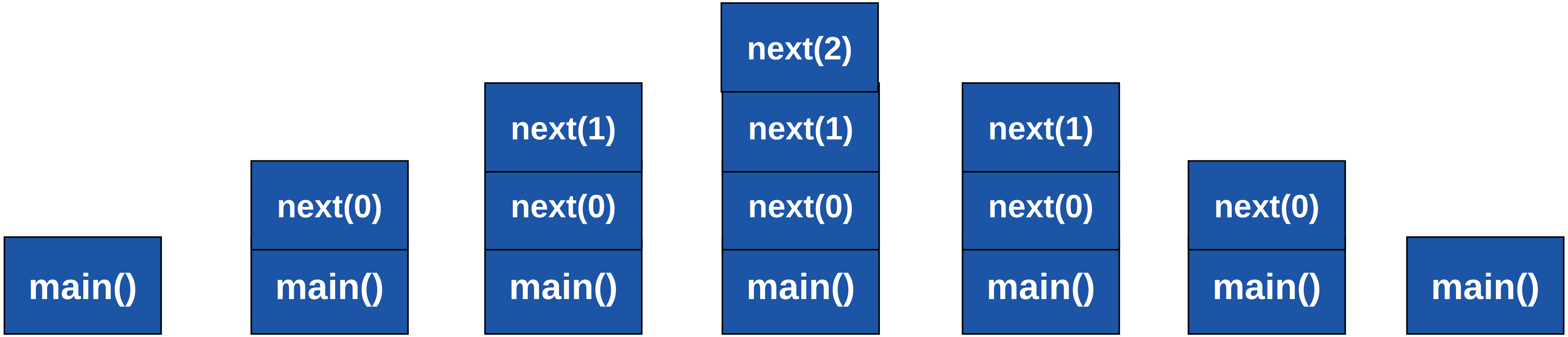
Recursion

A simple example

```
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);     recursion (a recursive call)
        } else {
            StdOut.println(" ready");     base case
        }
 }
}
```

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)

Base case the condition is “false”, else is executed and next(2) is returned.

Recursion

- Consider the following sequence of numbers:

1, 3, 6, 10, 15....

- Design a program which calculates the Nth number in the sequence using:
  1. for-loop
  2. while-loop
  3. recursion

for-loop ascending

```
int triangle(int n) {
    int sum= 0;
    for (int i=0; i<=n; i++) {
        sum = sum + i;
    }
    return sum;
}
```

while-loop descending

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

Now, the same problem solved recursively

Recursive solution (1)

```
int triangle(int n) {
    if (n == 1){
      return 1;   ← Base case
    } else {
       return (triangle(n-1) + n);   ← recursion
    }
}
```

Recursive solution (2)

```
int triangle(int n) {
    if (n == 1){
      return 1;        Base case
    } else {
       return (n + triangle(n-1));     recursion
    }
}
```

Tail recursion

- It is possible to optimize the usage of the execution stack when the last thing in the method is the recursive call.
  - If the last thing the method does is a recursive call – then there is no need to add another activation record on the stack . This is called tail recusion optimization.

  ⇒ The execution stack will no longer grow with the number of recursive calls

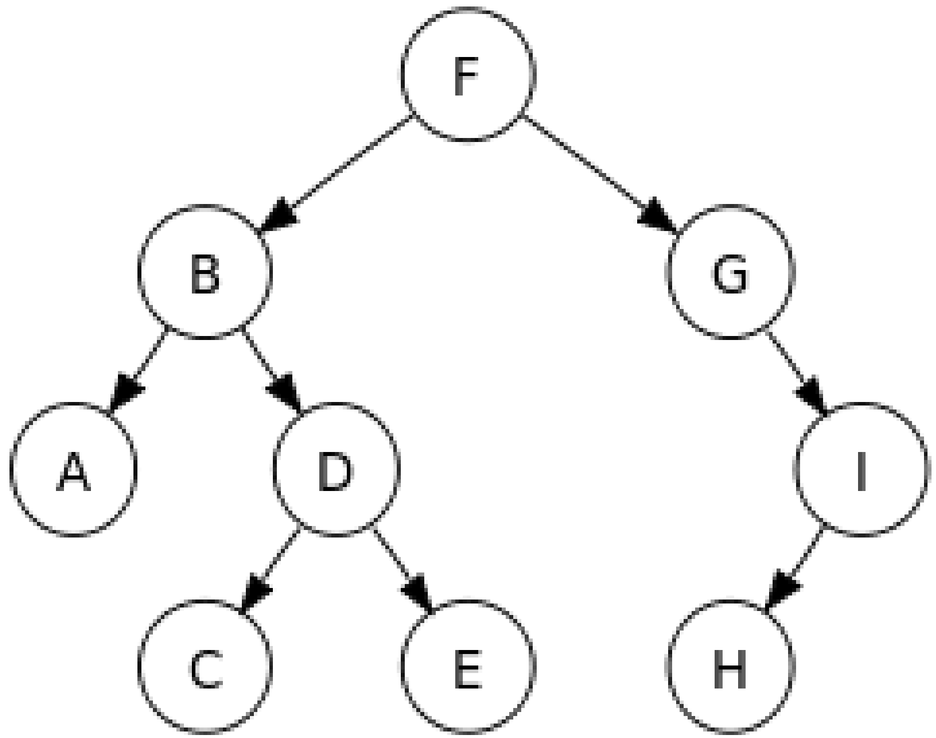
```
int triangle(int n) {
    if (n == 1){
      return 1;
    } else {
       return (n + triangle(n-1));
    }
// No more operations in the method
}
```

The last operation is a recursive call

If one or more statements are added after the recursive call it is no longer possible to do tail recursion optimization

Recursion – Why?

- Some problems in computer science are easier solved by recursion:
  - Traversing a file system (a tree)
  - Traversing a binary search tree.



Factorial

- Defined as:

  n! = n * (n-1) * (n-2) .... * 1;

- Eg.:

  1! = 1 (base case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

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

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

- Divide and conquer approach:
  1. What can be sollved in one statement (base case)
  2. Solve the remainder of the problem (i.e. with many similar statements)
     - Resuse the method to solve the remainder – recursive call

Factorial programs

```
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));
    }
}
```

Fibonacci numbers

- Fibonacci numbers
  - Each number in the sequence is the sum of the two preceeding numbers
    - eg., 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) and fibonacci(1) are the base cases

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

Recursion vs. Loops

- Recursion
  - Based on conditional statements (`if`, `if…else` or `switch`)
  - Repetition by successive method calls
  - Terminates when the base case(s) are reached/true
  - Successive recursive calls should deal with a smaller partition of the problem (non-overlapping) for the recursion to terminate
- Loops
  - Builds on `for`, `while` or `do…while`
  - Repetition by explicit representation of a block of code
  - Terminates when the loop-condition is false or when “break” is executed.
  - Controls repetition by counters

Recursion vs. Loops (cont.)

- Recursion
  - More overhead compared to iteration
    - One exception: some systems can do efficient tail recursion optimization
  - Use more stack space
    - One exception: some systems can do efficient tail recursion optimization
  - Recursive problems can be solved by loops
  - Often generate compact (easy to understand) code

Recursion summary

- Basic idea: reduce the problem to a simpler problem that can be solved by the same method/has the same structure

- Base case: the recursion must reach a case in which no more recursive calls are made (base case) for the recursion to terminate