Slides adapted from Algorithms 4th Edition, Sedgewick.

ID1020 Algorithms and Data Structures

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

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

Dr. Per Brand
perbrand52@gmail.com

Course Overview

- Programming and solving problems with applications
- Algorithm: a method for solving a problem
- Data Structure: a structure where information is stored
- Evaluating algorithms: primarily computational complexity

Area	Chapter in Book	Algorithms and Data Structures
Data Types	1	stack, queue, bag, union-find
Sorting	2	quicksort, mergesort, heapsort, priority queue
Search	3	BST, red-black BST, hashtabell
Graphs	4	BFS, DFS, Prim, Kruskal, Dijkstra

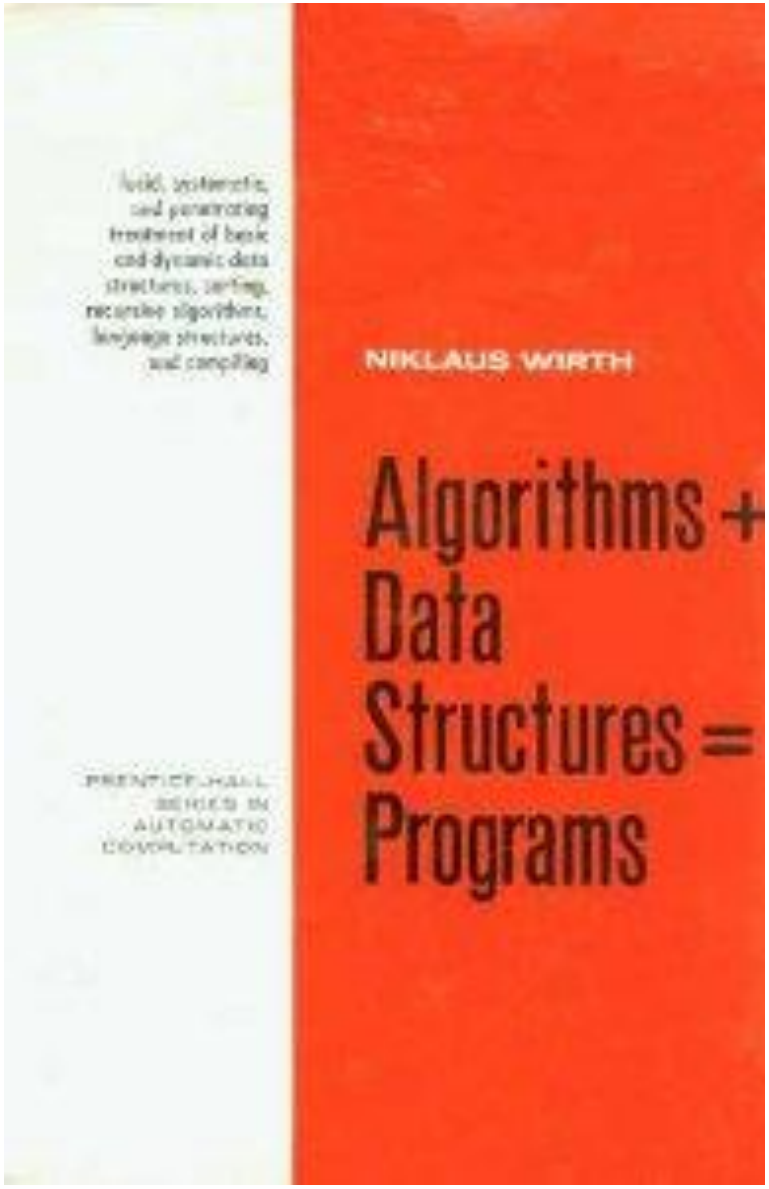
Why study Algorithms and Data Structures?

- Large impact on the world.

  - Internet. Web search, computer networks, browsers, ...
  - Computers. Hardware, graphics, games, storage, ...
  - Security. Cell phones, e-commerce, voting machines, ...
  - Biology. Whole Genome Sequencing, protein folding,..
  - Social Networks. Recommendations, newsfeeds, ads, ...
  - Physics. Large Halydron Collider – Higgs Boson, N-body simulation, particle collision simulation, ...

To be able to build Software

- Algorithms + Data Structures = Programs
  - Niklaus Wirth, 1976



To become a better software developer

- "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)

Data-Driven Science

- Computational models are starting to replace analytic models in the natural sciences.

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

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

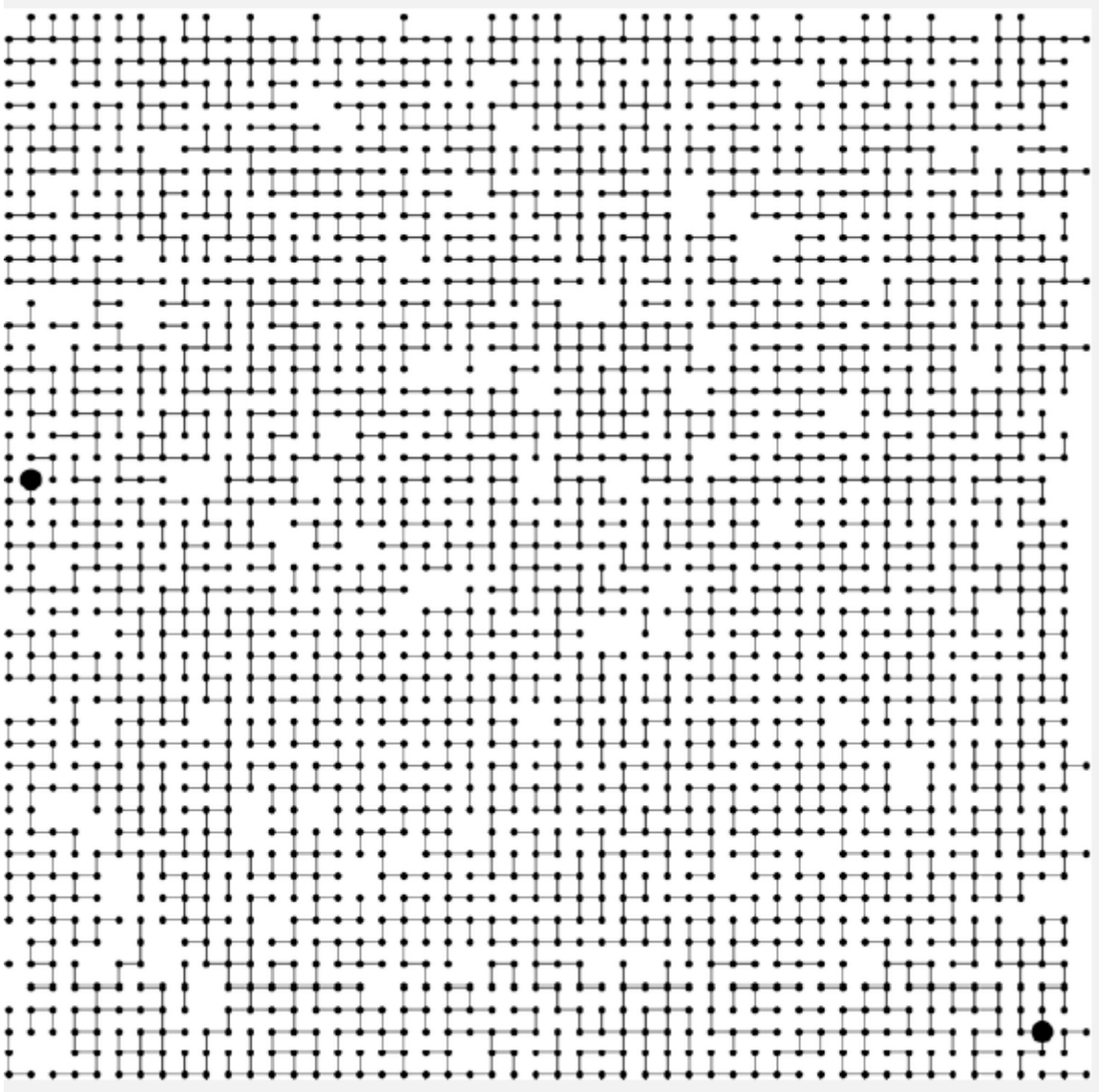
1900-talet (equation-based)

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

2000-talet (algorithm-based)

To solve difficult problems…

For example,
network connectivity



Why study algorithms?

- Because it pays to do so!

Prerequisites

- A completed course in programming (t.ex. ID1018)
  - loops, arrays, funktions, object-oriented programming, Strings
- Experience programming in Java

Resources

- Bilda
  - All coursework assignments will be submitted through Bilda
- No group work
  - The coursework and exam are examined individually
- Home page
  - www.it.kth.se/courses/ID1020

More Course Information

- Requirements
  - pass the coursework
  - exam

- Lab assistants
  - **Kamal Hakimzadeh**
  - **Mahmoud Ismail**
  - Lars Kroll
  - Alex Ormenisan

- **OBLIGATORY REGISTRATION FOR THE EXAM VIA DAISY**

The Book

- GET AND READ THE BOOK!
  - **Algorithms 4th Edition, Sedgewick.**

- Tips
  - Start with the labs and projects in good time!
  - Do the homework on Bilda – they are not too challenging and help prepare you for the exam

Examination

- Homework (10%)
  - 4th och 11th November. Submit using Bilda.
- 5 labs (50%)
  - 8th, 18th (submit on Bilda), 23rd November
  - 29th, 12th December
- 2 programming projects (40%)
  - 6th December
  - 9th January

Course work
(4.5 Credits)

- Exam (3.0 Credits)
  - 11th January

Programming Models and Data Abstraction

- Recursion (chapter 1.1, page 25 in the book)
  - A method can call itself!

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

Basic Java constructs

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

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

Basic Java constructs

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

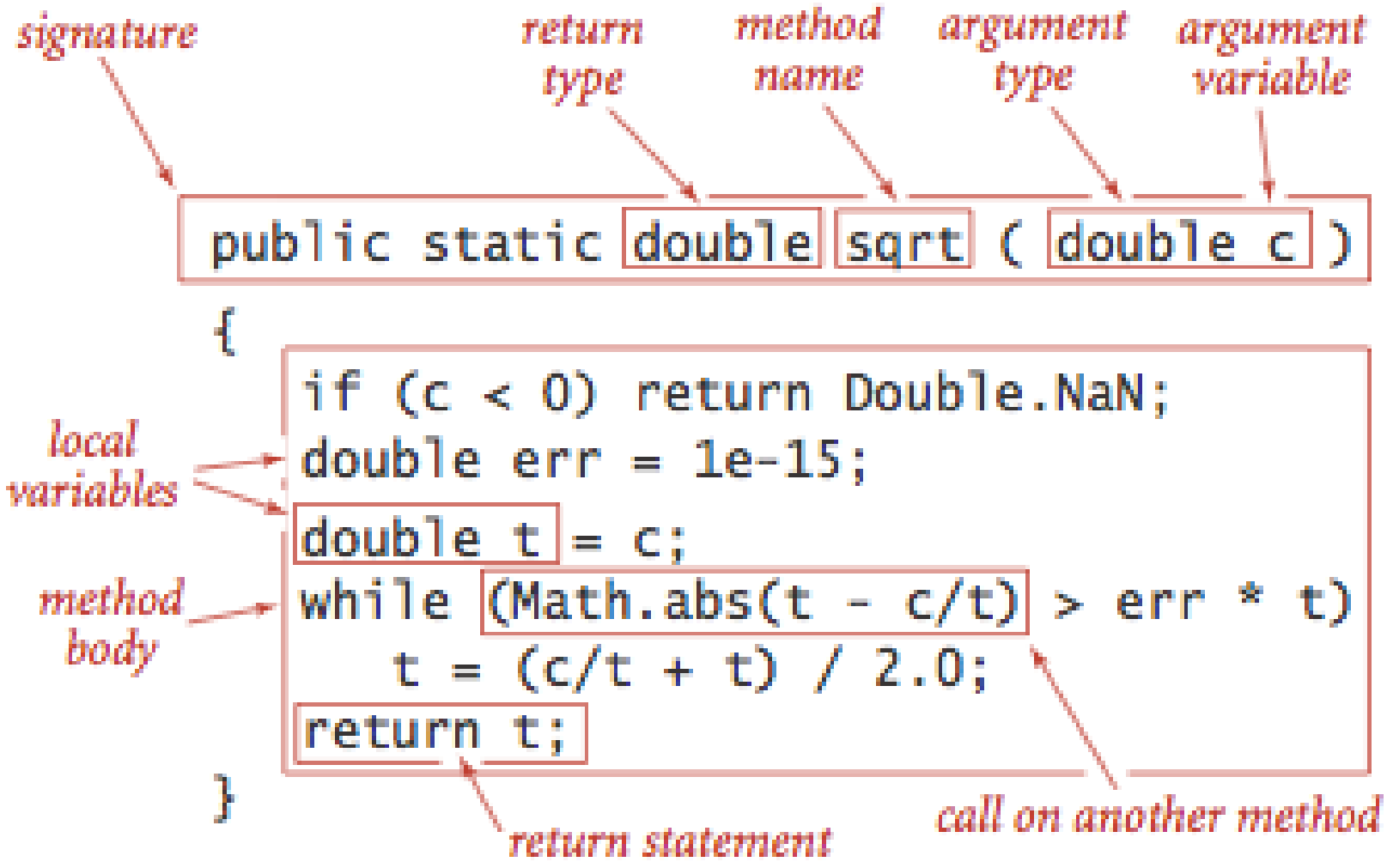
```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

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

Aliasing

```
int[] a = new int[N];
  ...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678; // what value does a[i] hold now?
```

Static Methods



Properties of Methods

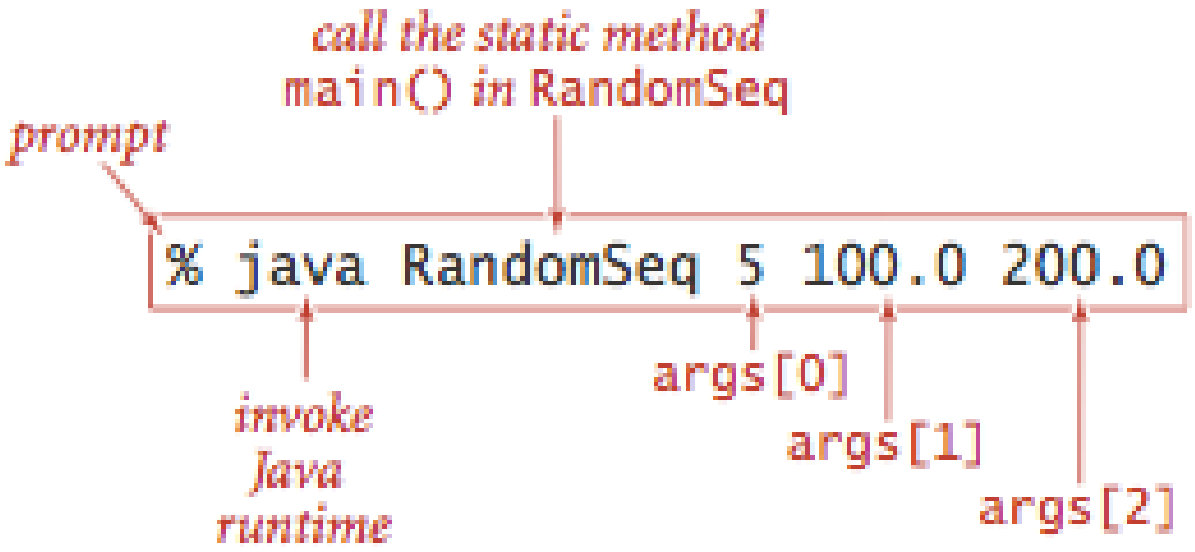
- Argument
  - *pass-by-value* (primitive types)
  - *pass-by-reference* (objects)
- Method names can be *overloaded*
  - *Math.min(int x, int y), Math.min(double x, double y)*
- Methods can have at most one return value, but can return from many different places in a method.
- A method can have *side-effects*
  - For example, update a member variable in an object

Input och Output in Java

- A Java program can receive input values from:
  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 values to:
  1. *standard-output stream* (stdout)

Executing a Java program



Formated 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 "

Redirecting from stdin (standard input)

redirecting from a file to standard input

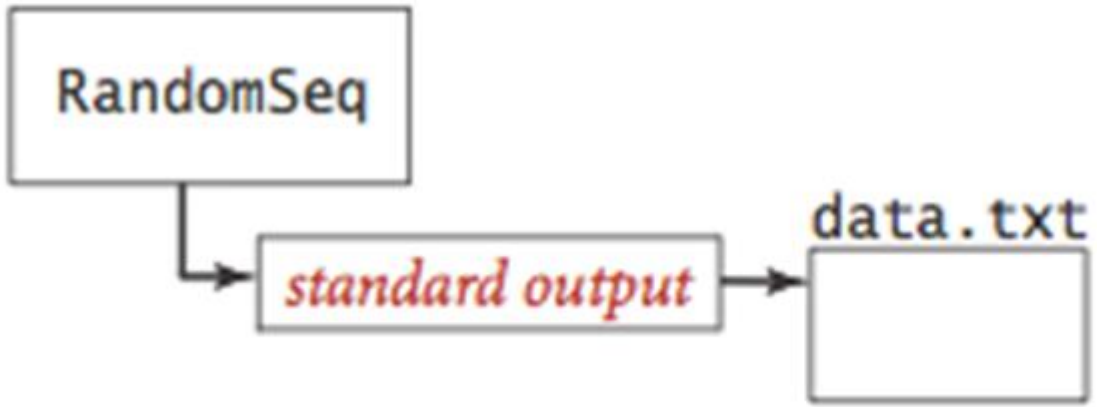
```
% java Average < data.txt
```

data.txt

Redirecting to stdout (standard out)

redirecting standard output to a file

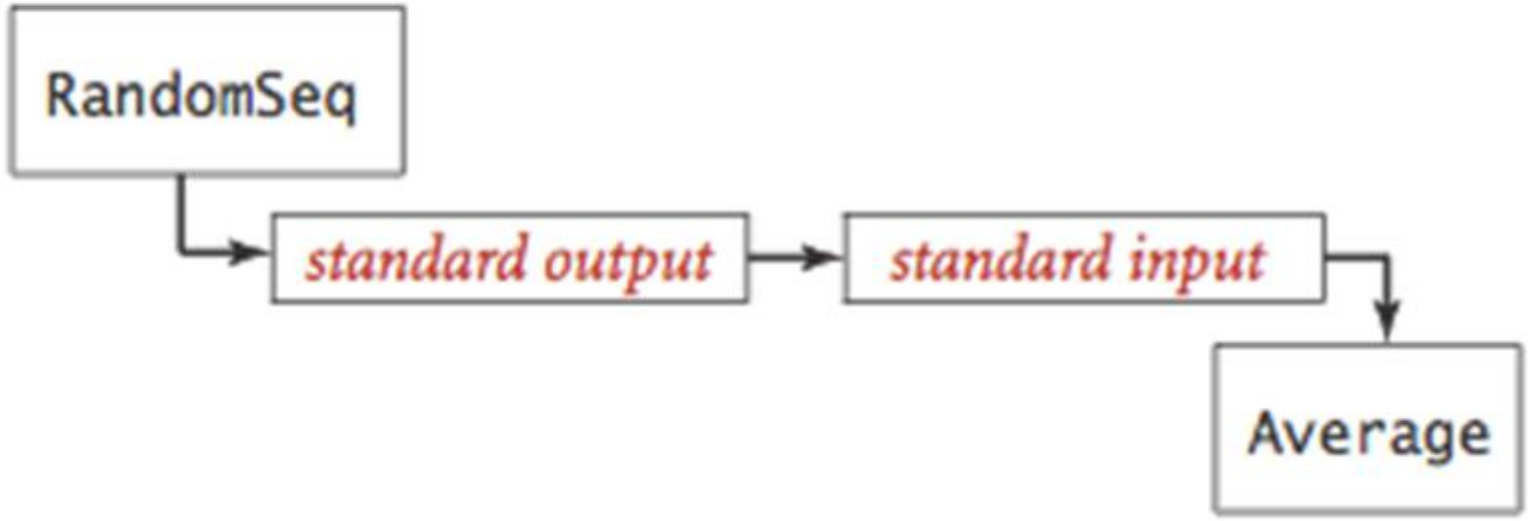
```
% java RandomSeq 1000 100.0 200.0 > data.txt
```



Piping output from one program to the input of another

piping the output of one program to the input of another

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



APIs and Object-Oriented Programming

Data abstraction

- Object-Oriented design
  - Abstract data types

- An *Application Programming Interface (***API***)* is an interface that specifies the behavior of an abstract data type (a contract).

- An API *encapsulates* the behavior of an abstract data type.
  - The API client knows nothing about how the internals (or implementation of) the abstract data type.

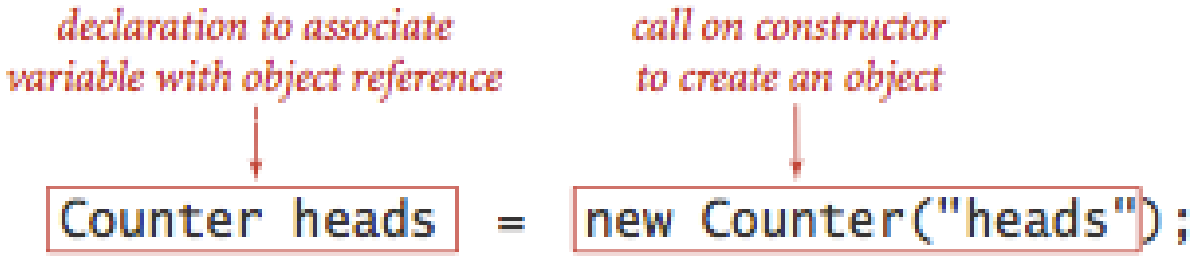
Classes and Objects

- Which operations can you see in the API for the Counter class?

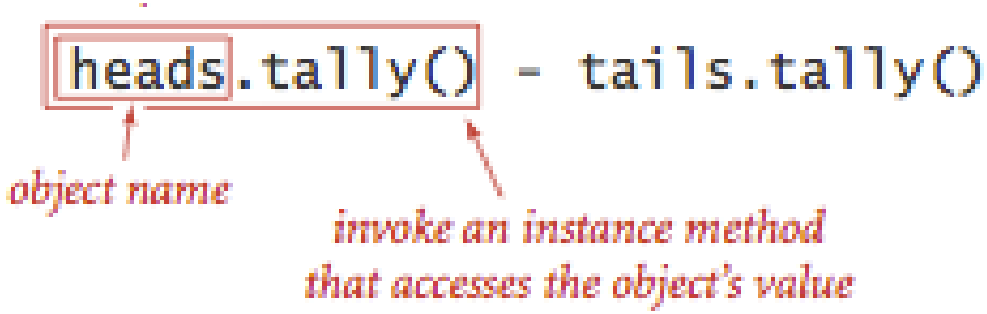
public class Counter		
	Counter(String id)	*create a counter named* id
void	increment()	*increment the counter by one*
int	tally()	*number of increments since creation*
String	toString()	*string representation*

Counter Classs API

- Creating objects



- Calling (invoking) methods



How do you write a good API?

- Encapsulation
- Clear contract
- Give the client what it needs, and no more.
- Bad properties of an API
  - Repetiton of methods
  - Too difficult to implement
  - Too difficult to use by the client
  - Too *narrow* – missing methods that the client needs
  - Too *wide* – includes methods the client doesn't need
  - T*oo general* – no useful abstractions
  - Too specific– abstractions help too few clients
  - Too tightly coupled to a representation – clients have to know details of the representation (*leaky abstraction*)

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*-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)

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

Recursion

A simple recursive program

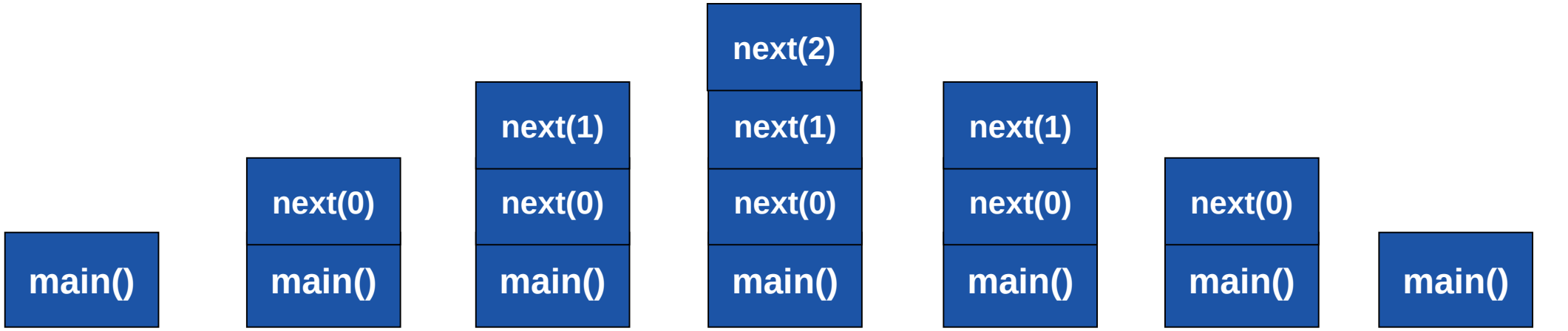
```
public class Recursion
{
  public static void main (String args[])
  {
        next(0);
  }
  public static void next(int index)
  {
        StdOut.print(index);
        if (index < 2) {
            next(index+1);
        } else {
            StdOut.println(" klar");
        }
  }
}
```

recursion here (a recursive call)

"base case" here

The program prints out:
012 klar

Visualize recursion with time as a "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: condition becomes "false", and the 'else condition' is executed and next(2) returns.

Recursion

- Consider the following series:

  1, 3, 6, 10, 15….

- Write a program that calculates the number N in the series:
  1. for-loop
  2. while-loop
  3. recursion

Find the n^{th} term (for-loop ascending)

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

Find the n^{th} term (while-loop descending)

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

Now, the same program with recursion

Find the n^{th} term (recursion)

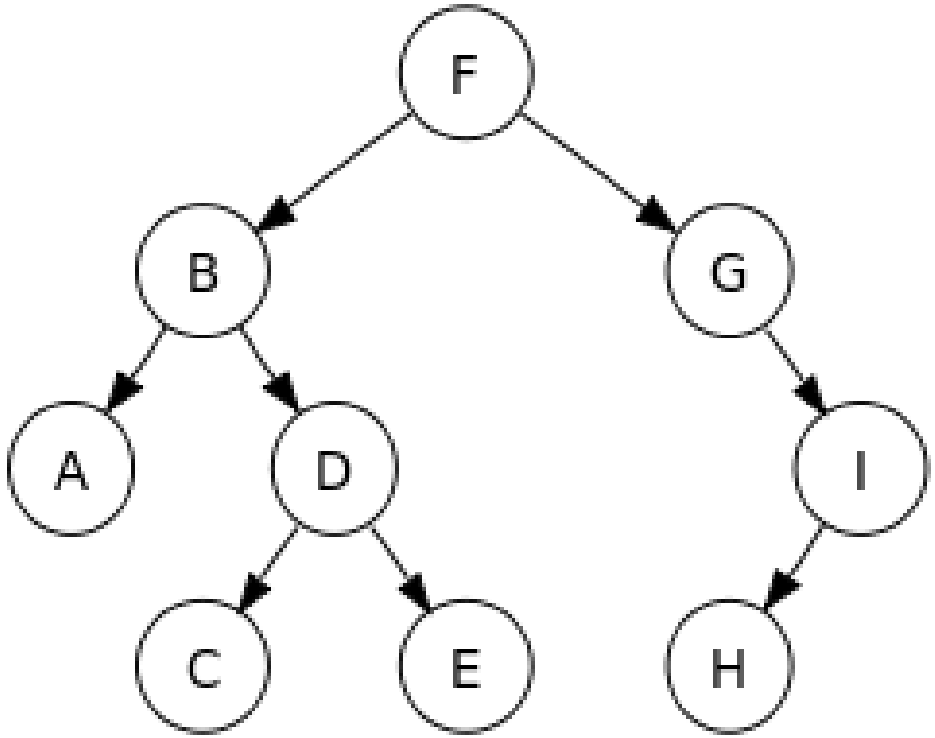
```
int triangle(int n) {
     if (n == 1){
       return 1;
     } else {
        return (n + triangle(n-1));
     }
}
```

base-case

recursive call

Recursion - motivation

- In computer science, certain problems are easier to solve with the help of a recursive function:
  - Traverse a file system.
  - Traverse a tree of search results.



Factorial

- Factorial is defined as:

  n! = n * (n-1) * (n-2) …. * 1;

- For example,

  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

- We will try and solve the problem in two parts:
  1. What the program can solve in one statement (base-case)
  2. What the program can solve with many such statements
     - Then we call a copy of the function again to do the next "step".

Factorial 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));
    }
}
```

The Fibonacci sequence

- The Fibonacci sequence
  - Every element is the sum of the two previous values in the Fibonacci sequence:

    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

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

Recursion and Loops

- Recursion
  - Builds on conditional statements (`if`, `if…else` or `switch`)
  - Repetition with the help of repeated method calls
  - Terminates when the base-case is true (or has been reached)
  - Controls repetition by subdividing the problem into several simpler problems
- Loops
  - Build with `for`, `while` or `do…while`
  - Repetition with the help of an explicit *repetition code-block*
  - Terminates when loop conditions become false or a "break" is called.
  - Controls repetition with the help of a counter

Recursion and Loops (ctd.)

- Recursion
  - More overhead than iteration
    - an exception is tail-recursion is certain implementations
  - Requires more stack memory
    - an exception is tail-recursion is certain implementations
  - Can be solved with loops
  - Can typically be written in fewer lines of source-code

Tail recursion

- An optimized type of recursion where the last operation in a function is a recursive call.
  - The call is the last statement in the function, so it can be replaced by a jump, as no return address needs to be saved on the stack. It's called tail-recursion optimization.

  ⇒ the stack no longer grows in proportion to the number of recursive calls made

```
int triangle(int n) {
    if (n == 1){
      return 1;
    } else {
       return (n + triangle(n-1));
    }
// no more statements in the function
}
```

last statement is a recursive call

If you add any more statements after the recursive call, it will no longer be tail-recursion.

Recursion summary

- Recurisve thinking: reduce the problem to a simpler problem with the same structure
- Base-case: there has to be a case that does not lead to a recursive call