

Elements of Computer Science Programming

Ralf Schenkel
based on slides by Norbert Müller

16. Oktober 2023

1 Introduction

Topic of this course:

Writing small programs

Planned content:

- Problem, Algorithm, Program
- Java as an example for a modern programming language
- Non-primitive data types in Java
- Classes and exceptions
- Exceptions
- Useful techniques: generics, collections, ...

Software: **Java Platform, Standard Edition ('SE')**

- all examples will be based on Java
- Most recent version: Java Platform, Standard Edition 21
`https://jdk.java.net/21/`
- We will only use Java features up to version 8
- OpenJDK Development Kit (JDK) + Java Documentation
- available on the computer science CIP pools (H523,H524)
- equivalent versions on Windows, Linux, ...
- Integrated development environment: Eclipse, IntelliJ, ...

further information on Java online:

- R. Gallardo, S. Hommel, S. Kannan, J. Gordon, S.B. Zakhour, 'The Java Tutorial',
most recent version: Java SE Tutorial 2022-03-04,
<http://download.oracle.com/javase/tutorial>
- 'The Java Language Specification, Java SE 21 Edition',
Gosling/Joy/Steele/Bracha/Buckley/Smith
(docs.oracle.com/javase/specs/)
- 'Java Look and Feel Design Guidelines', 2nd ed., Addison Wesley, 2001.
<http://www.oracle.com/technetwork/java/jl1f-135985.html>
- 'The Java Virtual Machine Specification, Java SE 21 Edition',
Lindholm/Yellin/Bracha/Buckley/Smith
(docs.oracle.com/javase/specs/)

Book on Java

- Quentin Charatan, Aaron Kans
Java in Two Semesters
Springer, 2019. <https://link.springer.com/book/10.1007/978-3-319-99420-8>

Readings on computer science in general

- Robert Sedgewick, Kevin Wayne
Computer Science: An Interdisciplinary Approach
Addison Wesley, 2017.

2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- Insertion: Java examples, part 1
- Flow control of algorithms
- Insertion: Java examples, part 2
- Structured composition of algorithms

- A **problem** is identified by an initial state ('problem exists' or 'problem identified') and by the aim to reach a target state ('problem solved').
- An **algorithm** is a precise, finite method for solving a problem, i.e., for the transition from 'problem identified' to 'problem solved'.
- A **program** is the encoding of an algorithm with a programming language for execution on a computer.

Specification of the problem:

- A precise algorithm requires a precise representation of the initial situation.
- We thus need to begin with an analysis of the problem in order to generate a specification that establishes the following:
 - ▶ information available in the initial situation.
(input data and their allowed values, i.e., their domain)
 - ▶ desired information.
(output data and their possible values, i.e., their range)
 - ▶ any conditions that need to be observed while deriving the desired information.
- The specification of the problem should be as detailed, as exact, and as complete as possible.

Examples for specifications of problems:

- **Given:** the three numbers 5, 7, and 11.
Wanted: their arithmetic mean.
- **Given:** flour, water, olive oil, salt.
Wanted: pizza dough made from these ingredients.
Conditions: One can use at most 400g flour and 2 spoons of olive oil.
- **Given:** place of residence Trier, holiday destination Tahiti, possible dates for vacation, set of flight schedules.
Wanted: proposals for trips.
Conditions: departure on Friday or Saturday, duration 4 weeks, cost not exceeding 3000 euros

2 Problem, Algorithm, Program

- What is a 'problem'?
- **What is an 'algorithm'?**
- What is a 'program'?
- Insertion: Java examples, part 1
- Flow control of algorithms
- Insertion: Java examples, part 2
- Structured composition of algorithms

The term **algorithm** dates back to the Arabic mathematician
Mohammed ibn Musa abu Djafar al Khowarizmi
(ca. 783 - 850) and his book

‘Kitab al muhtasar fi hisab al gebr we al muqabala’
(‘The Compendious Book on Calculation
by Completion and Balancing’).

In the Latin translation each section started with ‘Dixit algorismi’ (‘thus said al Khowarizmi’), which led to the later term algorithm.

What is an 'algorithm'?

We need to clarify the following questions:

- Which processing steps are required in an algorithm?
- How can we control the flow of the processing steps?
- How can we represent and write down algorithms?
- Which properties of algorithms are interesting?

Intuitive algorithms

Examples from everyday life:

process	algorithm	example for processing step
knit a pullover	knitting pattern	purl – plain
construct a model car	assembly instructions	glue outside mirror to left door
make a cake	recipe	beat 3 eggs until fluffy
play music piece	sheet of music	play c”

Such daily operation procedures are often not exact and leave room for interpretations, so they are not algorithms.

Example: starting with a car from the right curb as **intuitive algorithm**

- ➊ fasten seat belt, check rear-view mirror, if necessary pull handbrake, insert idle.
- ➋ briefly actuate ignition.
- ➌ If engine does not start, repeat step 2.
- ➍ look around and wait for a traffic gap
- ➎ pedal the clutch with left foot and brake with right foot, enable the left indicator, release the handbrake.
- ➏ hold clutch, engage 1st gear.
- ➐ accelerate with right foot, turn the steering wheel to the left.
- ➑ release the clutch, accelerate, turn the steering wheel such that the car reaches the right lane.
- ➒ ...

Observations from the examples

- The individual steps are executed by a processor.
- A single processor executes one step after the other.
- Usually a step can only be executed after the previous step was successfully completed.
- The execution of a step can depend on a condition (*if a sufficiently large gap in the traffic exists then drive off, otherwise continue waiting*).
- Some steps must be repeated multiple times (*repeatedly briefly actuate ignition*).
- Repetitions always end when a certain condition gets true (*... until the engine starts*).

- In some cases the order in which some steps are executed does not matter. Thus transposition and parallel execution are possible (*pedal the clutch with left foot and brake with right foot*)
- In principle an algorithm executes a transformation from an initial state to a final state. Or: for a given input the application of an algorithm creates a well-defined output.
- In a mathematical sense an algorithm defines a map $f : E \rightarrow A$ from the set of input data E to the set of output data A .
- Usually the algorithm cannot be applied in every initial state.

2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- **What is a 'program'?**
- Insertion: Java examples, part 1
- Flow control of algorithms
- Insertion: Java examples, part 2
- Structured composition of algorithms

program = formal representation of an algorithm in a specialized language (**programming language**)

Example: Java

```
1 public class K2B01E_first_Example {  
2     public static void main(String[] args) {  
3  
4         int index;  
5         int count;  
6  
7         index = 0;  
8         count = 10;  
9  
10        while ( index <= count )  
11        {  
12            System.out.println( index * index );  
13            index = index + 1;  
14        }  
15    }  
16 }
```

2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- **Insertion: Java examples, part 1**
- Flow control of algorithms
- Insertion: Java examples, part 2
- Structured composition of algorithms

A first Java example

```
1 public class K2B01E_first_Example {
2     public static void main(String[] args) {
3
4         int index;
5         int count;
6
7         index = 0;
8         count = 10;
9
10        while ( index <= count )
11        {
12            System.out.println( index * index );
13            index = index + 1;
14        }
15    }
16 }
```

Typical components:

- separation symbols like `;`, `,`, parenthesis like `()`, `[]`, `{}`
- header (lines 1-2), declarations (lines 4-5), assignments (lines 7-8,13), loops (lines 10-14)
- method calls (line 12)

Typical sequence for writing a program:

- 1 Write the program with a 'simple' ASCII editor
 - ▶ Text editors like MS Word, LibreOffice Write etc. are unsuitable!
 - ▶ alternative with MS Windows: Notepad , Notepad++ , ...
 - ▶ with Linux: kwrite , nano , vi , gedit , geany , ...
 - ▶ or use an integrated development environment, e.g., Java-Editor , Eclipse , IntelliJ , ...
- 2 file name: K2B01E_first_Example.java (as in the header)
- 3 translate and start in the console:

```
javac K2B01E_first_Example.java  
java K2B01E_first_Example
```

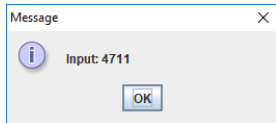
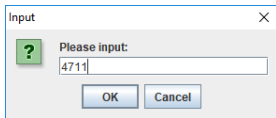
Example: user input/output via the console

```
1 public class K2B02E_IO_Console {  
2     public static void main(String[] args){  
3         int    inputNumber;  
4         String inputText;  
5  
6         inputText = System.console().readLine("Input: ");  
7  
8         inputNumber = Integer.parseInt(inputText);  
9  
10        System.out.println("Output: " + inputNumber);  
11    }  
12 }
```

```
1 > javac K2B02E_IO_Console.java  
2 > java  K2B02E_IO_Console  
3 Input: 1243  
4 Output: 1243
```

Example: user input/output with popup windows

```
1 import javax.swing.JOptionPane;
2
3 public class K2B03E_IO_Dialogue {
4     public static void main(String[] args) {
5         int    inputNumber;
6         String inputText;
7
8         inputText=JOptionPane.showInputDialog(null,"Please input:");
9
10        inputNumber = Integer.parseInt(inputText);
11
12        JOptionPane.showMessageDialog (null, "Input: "
13                                       + inputNumber);
14    }
15 }
```



Example: user input as arguments to the program

```
1 public class K2B04E_IO_Parameters {  
2     public static void main(String[] args) {  
3         int    inputNumber;  
4         String inputText;  
5  
6         inputText=args[0];  
7  
8         inputNumber = Integer.parseInt(inputText);  
9  
10        System.out.println("Input: " + inputNumber);  
11    }  
12 }
```

- as a standalone program in the console:

```
1 javac K2B04E_IO_Parameters.java  
2 java  K2B04E_IO_Parameters 4711
```

- IDE: in **Eclipse** via **Run** → **Run Configuration**, then configure under **Arguments**

Example: user input with 'Scanner'

```
1 import java.util.Scanner;
2
3 public class K2B05E_IO_Scanner {
4     public static void main(String[] args){
5         Scanner sc = new Scanner(System.in);
6
7         int inputNumber;
8
9         System.out.println("Input: ");
10
11         inputNumber = sc.nextInt();
12
13         System.out.println("Output: " + inputNumber);
14     }
15 }
```

```
1 > javac K2B05E_IO_Scanner.java
2 > java K2B05E_IO_Scanner
3 Input:
4 1243
5 Output: 1243
```

Example: arrays for storing multiple numbers

```
1 public class K2B06E_Arrays {
2     public static void main(String[] args) {
3         int index;
4         int count;
5
6         int[] array;
7
8         count=5;
9         array = new int[count];
10
11         index=0;
12         while (index<count)
13         {
14             array[index] = index*index;
15             index=index+1;
16         }
17
18         index=0;
19         while (index<count)
20         {
21             System.out.println(array[index]);
22             if (index<count) System.out.println("-");
23             index=index+1;
24         }
25     }
26 }
```

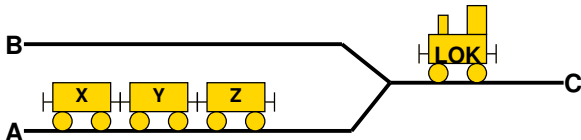
2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- Insertion: Java examples, part 1
- **Flow control of algorithms**
- Insertion: Java examples, part 2
- Structured composition of algorithms

informal example: shunting algorithm

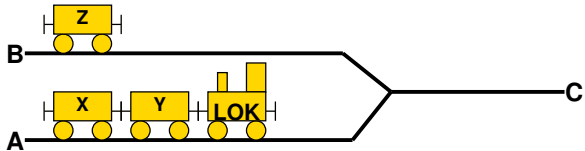
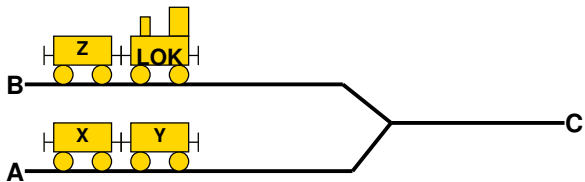
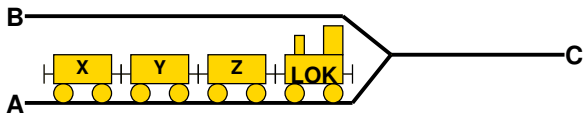
- given:

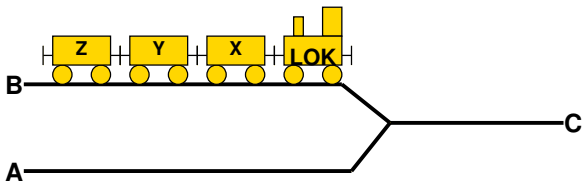
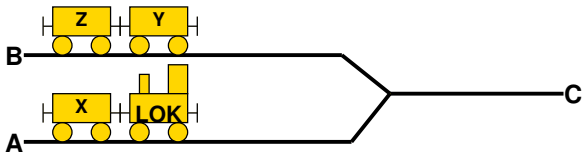
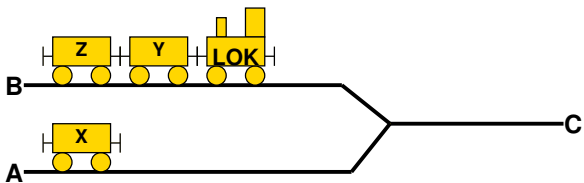
- 1 railway tracks with 2 storage tracks **A**, **B** and a track **C** with switch to **A** and **B**.
- 2 coaches in order **X**, **Y**, **Z** on track **A**
- 3 shunting engine **LOK** on track **C**



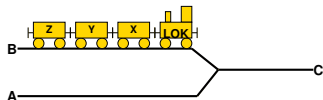
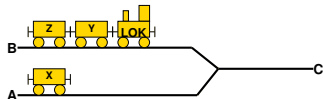
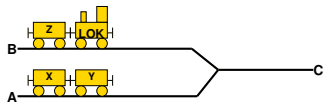
- wanted:

- ▶ Algorithm for reordering the coaches using the engine, target order **Z**, **Y**, **X** on track **B**.





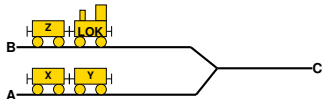
- 01 Move engine to track **A** and couple coach.
- 02 Uncouple next coach.
- 03 Move engine and coach to track **C**.
- 04 Continue to track **B**.
- 05 Uncouple engine.
- 06 Move to track **C**.
- 07 Move engine to track **A** and couple coach.
- 08 Uncouple next coach.
- 09 Move engine and coach to track **C**.
- 10 Continue to track **B**.
- 11 Couple coaches and uncouple engine.
- 12 Move to track **C**.
- 13 Move engine to track **A** and couple coach.
- 14 (no need to uncouple anything)
- 15 Move engine and coach to track **C**.
- 16 Continue to track **B**.
- 17 Couple coaches and uncouple engine.
- 18 Move to track **C**.



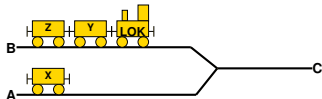
Observation 1:

Actions are done consecutively (in sequential order)

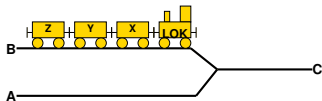
- 01 Move engine to track **A** and couple coach.
- 02 Uncouple next coach.
- 03 Move engine and coach to track **C**.
- 04 Continue to track **B**.
- 05 Uncouple engine.
- 06 Move to track **C**.



- 07 Move engine to track **A** and couple coach.
- 08 Uncouple next coach.
- 09 Move engine and coach to track **C**.
- 10 Continue to track **B**.
- 11 Couple coaches and uncouple engine.
- 12 Move to track **C**.



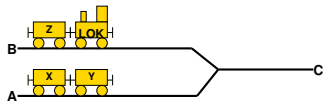
- 13 Move engine to track **A** and couple coach.
- 14 (no need to uncouple anything)
- 15 Move engine and coach to track **C**.
- 16 Continue to track **B**.
- 17 Couple coaches and uncouple engine.
- 18 Move to track **C**.



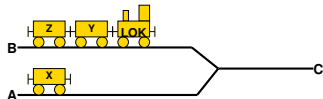
Observation 2:

Some sequences of actions are repeated

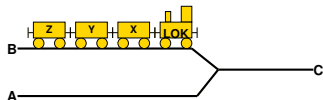
01 Move engine to track **A** and couple coach.
 02 **Uncouple next coach.**
 03 Move engine and coach to track **C**.
 04 Continue to track **B**.
 05 **Uncouple engine.**
 06 Move to track **C**.



07 Move engine to track **A** and couple coach.
 08 **Uncouple next coach.**
 09 Move engine and coach to track **C**.
 10 Continue to track **B**.
 11 **Couple coaches and uncouple engine.**
 12 Move to track **C**.



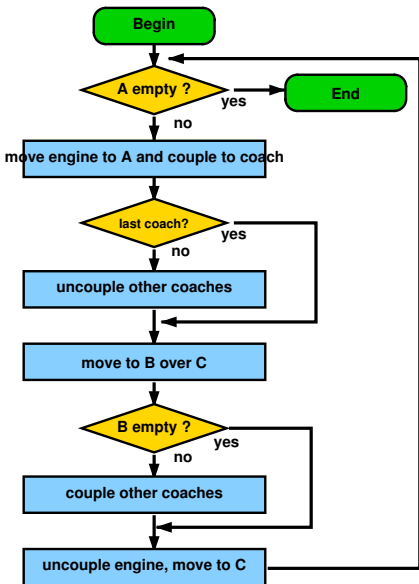
13 Move engine to track **A** and couple coach.
 14 **(no need to uncouple anything)**
 15 Move engine and coach to track **C**.
 16 Continue to track **B**.
 17 **Couple coaches and uncouple engine.**
 18 Move to track **C**.



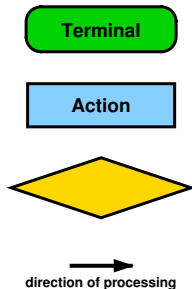
Observation 3:

Under some conditions actions are chosen from several alternatives

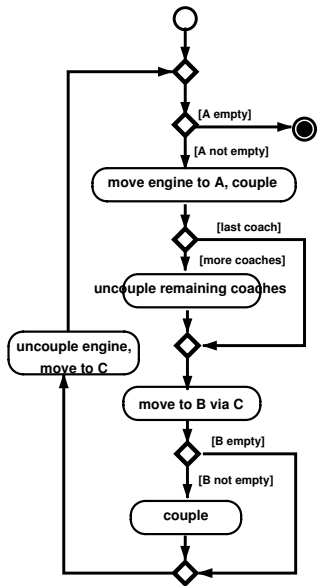
Flow diagrams to represent algorithms



flow diagrams:
established method
to represent
Algorithms



UML for representing algorithms



Unified Modeling Language (UML):
standardized graphical modelling language
for specifying, constructing and
documenting (software) systems

Begin



End



Direction



Action

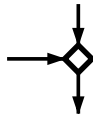
Decision



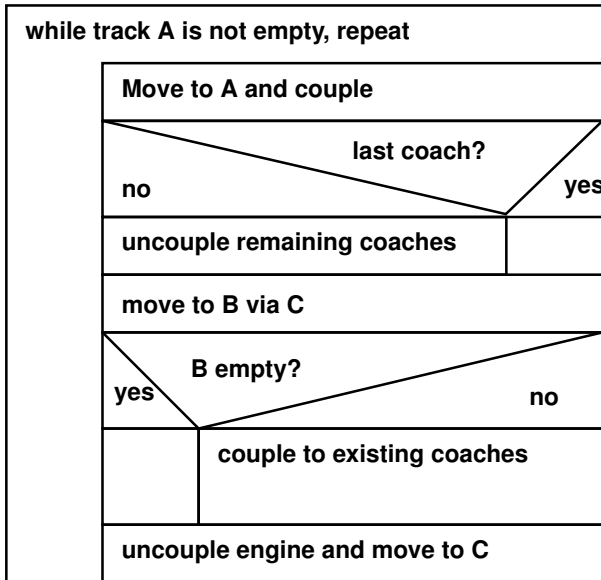
[Condition 1]

[Condition 2]

Join



Structograms for representing algorithms



2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- Insertion: Java examples, part 1
- Flow control of algorithms
- **Insertion: Java examples, part 2**
- Structured composition of algorithms

Shunting as a Java program

Input the desired length of the track,
then create arrays of corresponding length **track_a**, **track_b**:

```
1 public class K2B07E_Shunting {
2     public static void main(String[] args) {
3         int index;
4         int count;
5
6         int[] track_a;
7         int[] track_b;
8
9         count = Integer.parseInt(
10             System.console().readLine("Count: "));
11
12         track_a = new int[count];
13         track_b = new int[count];
14     }
```

Initialize **track_a**,
then output:

```
1    index=0;
2    while ( index < count )
3    {
4        track_a[index] = index*index;
5        index=index+1;
6    }
7
8    index=0;
9    while ( index < count )
10   {
11       System.out.print(track_a[index]);
12       index=index+1;
13       if (index<count){
14           System.out.print("-");
15       } else {
16           System.out.println();
17       }
18   }
```


Assign **track_b** in reversed order,
then output again:

```
1    index=0;
2    while ( index < count )
3    {
4        track_b[index]=track_a[count-1-index];
5        index=index+1;
6    }
7
8    index=0;
9    while ( index < count )
10   {
11       System.out.print(track_b[index]);
12       index=index+1;
13       if (index<count){
14           System.out.print("-");
15       } else {
16           System.out.println();
17       }
18   }
19 }
20 }
```

2 Problem, Algorithm, Program

- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- Insertion: Java examples, part 1
- Flow control of algorithms
- Insertion: Java examples, part 2
- **Structured composition of algorithms**

Controlling the flow of algorithms

The order in which the instructions of an algorithm are executed follows three basic patterns:

- **Sequence**
- **Selection** (alternative, branching instruction)
- **Iteration** (repetition, loop)

The three concepts sequence, selection and iteration are enough to formulate *all* executable algorithms.

(\leadsto theory of computing)

Sequence:

- At every time only one action is executed.
- Every instruction is executed exactly once.
- The order of the execution corresponds to the given sequence of instructions.
- The sequence ends once all instructions were executed.
- Every sequence consists of at least one instruction.

The sequence is the basic module of every algorithm, i.e., every algorithm is a sequence of instructions.

Selection (alternative, branching instruction):

- general form (if-else-condition):

*if condition then **sequence1** else **sequence2***

If the logical expression **condition** evaluates to '**true**', then **sequence1** is executed, otherwise **sequence2**.

- special case (if-condition):

*if **Condition** then **sequence***

Corresponds to an if-else-condition with a **sequence2** which consists only of an 'empty' instruction.

Every instruction within a sequence can be a selection, i.e., selections can be 'nested'.

Iteration (repetition, loop):

- while-loop:

<i>while</i> condition <i>execute</i> sequence
--

- (1) First the logical expression **condition** is evaluated.
- (2a) If the evaluation yields the result '**true**',
sequence is executed.
Then the execution continues at (1).
- (2b) If the evaluation yields the result '**false**',
the repetition stops.

Every instruction within a sequence can be an iteration, i.e., iterations can be 'nested'.

- Sequence, selection, and iteration define **sequential algorithms**.
- If the flow control is extended by means to split (**fork**) and recombine (**join**) sequences executed in parallel, then **parallel algorithms** are possible.
- Sometimes parallel algorithms allow for a faster solution of problems, but they are not more powerful than sequential algorithms, i.e., they can solve the same problems.

Typical properties of algorithms

- **Finiteness:** The description of an algorithm has a finite length. During processing an algorithm, the created data structures and intermediate results are finite.
- **Termination:** The algorithm produces a result after a finite number of steps.
- **Abstraction:** An algorithm solves a class of problems; the actual problem instance to solve is determined by the input data.
- **Determination:** Algorithms are usually determined, i.e., they always return the same output values for the same input values each time they are executed.
- **Determinism:** An algorithm is deterministic, if there is at most one possible continuation at each step of its execution.

Structured Algorithms

An algorithm is **structured** if it is constructed following this ‘grammar’:

algorithm	←	sequence
sequence	←	instruction instruction sequence
instruction	←	simple_instruction selection_from_sequences iteration_of_a_sequence
simple_instruction	←	...

(Example how to read this: a sequence consists of (‘←’) an instruction or (‘|’) an instruction, followed by a sequence).

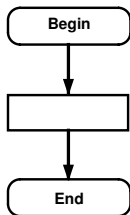
In a structured algorithm every instruction has exactly one entrance and one exit.

(goal: avoid intransparent processes, no ‘spaghetti code’)

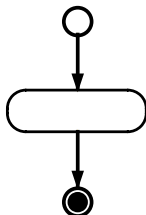
Construction of structured algorithms by 'refinement'

For a top-down construction of an algorithm we start with the most simple diagrams:

flow diagram



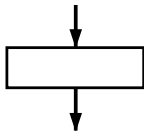
UML activity diagram



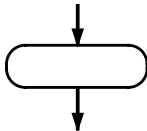
Goal: every structure has exactly one entrance and exactly one exit...

A sequence can consist of a sequence of instructions:

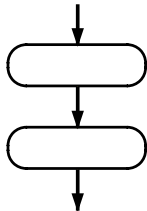
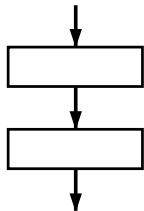
flow diagram



UML activity diagram

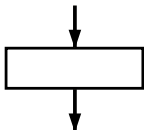


is refined as

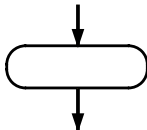


An instruction can consist of an iteration:

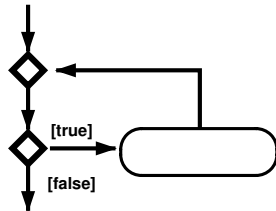
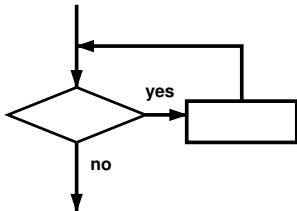
flow diagram



UML activity diagram

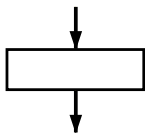


is refined as

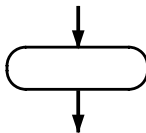


An instruction can consist of a selection:

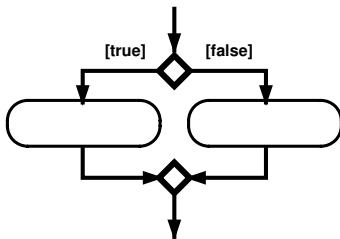
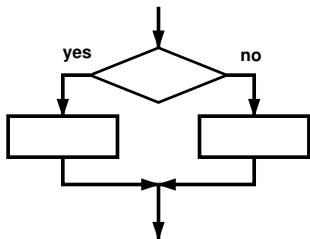
flow diagram



UML activity diagram

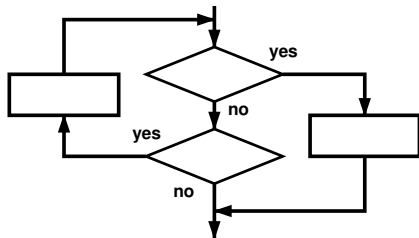


is refined as



Not every flow diagram or UML activity diagram is structured!

For example the following diagram cannot be constructed as shown before:



- The restriction to structured algorithms restricts the options to construct algorithms in favour of a better readability!
- For every flow diagram there is a structured flow diagram that computes the same result (see lectures on theory of computing).

Java as an example of a simple programming language

- Programming languages
 - Lexical structure of Java programs
 - Variable – Name – Value – Type
 - Literals and Constants
 - Truth values
 - Assignments and Expressions
 - Basic control structures in Java
 - Integer Numbers
 - Floating Point Numbers
 - Characters and Strings
 - Additional Operators
 - Methods

- A **program** is the implementation of an algorithm with a programming language in order to execute it on a computer.
- Very simple languages are in principle (!) sufficient to implement all computable algorithms.
Unfortunately, then the programming is very laborious.
- To make the implementation of specific types of algorithms more comfortable, a large number of programming languages have been developed.
- Thus programming languages are usually optimized for a specific scope.

Programming languages often follow one of the common basic patterns of programming (**programming paradigms**):

- imperative (Fortran, Cobol, Algol, Pascal, Modula, C, Java,...)
- object oriented (Smalltalk, C++, Delphi, Java, C#,...)
- functional (Lisp, OCAML, Haskell...)
- logical (Prolog,...)

More recent languages are often extended by components of other paradigms (e.g., Java with functional components).

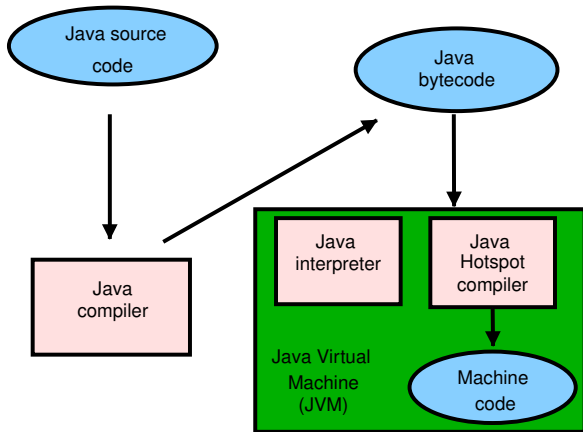
Java as an example of a current programming language:

- Java is: imperative, object oriented, platform independent, supported by libraries, Internet enabled
- The core of Java is relatively simple.
- In order to provide independence of the actual processor and operating system, a large number of ready program components is provided.

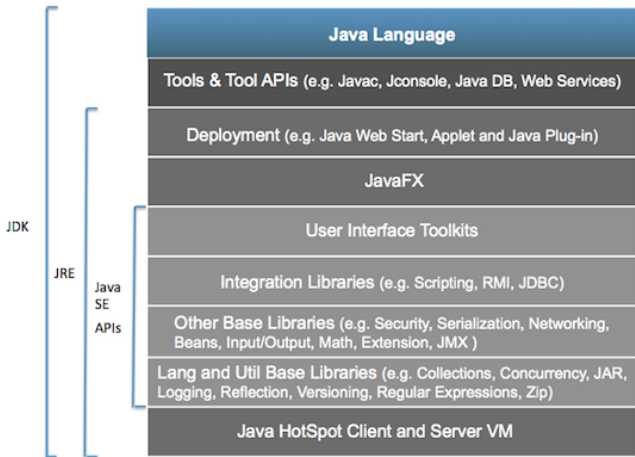


Java version	year	packages	classes/interfaces	members
1.0	1996	8	212	2125
1.1	1997	23	504	5478
1.2	1998	59	1520	ca. 16000
1.3	2000	76	1842	ca. 20000
1.4	2002	135	2991	ca. 32000
5.0	2004	165	>3000	
SE 6	2006	?	?	?
SE 7	2011	> 200	> 4000	?
SE 8 (LTS)	2014	> 200	> 4000	?
SE 9	2017		> 6000	
SE 10	3/2018			
SE 11	9/2018			
SE 12	3/2019			

Java uses source code and byte code that is machine-independent:



Conceptual architecture of Java SE 7:



Java as an example of a simple programming language

- Programming languages
- **Lexical structure of Java programs**
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

Elements of a simple Java program

```
1 import javax.swing.JOptionPane;
2 /* Greatest common divisor */
3 class K3B01E_GCD {
4     public static void main(String[] args) {
5
6         int a,b;
7
8         a = Integer.parseInt(JOptionPane.showInputDialog ("A=_"));
9         b = Integer.parseInt(JOptionPane.showInputDialog ("B=_"));
10
11        while (a != b){
12            if (a > b) a = a - b;
13            else b = b - a;
14        }
15
16        JOptionPane.showMessageDialog (null, "The GCD is: " + a);
17    }
18 }
```

- line 1: **package import**
- line 2: **comment**
pause
- line 3: **class header**
- line 4: **method header**
- line 6: **declarations**
- lines 8-9: **input**
- lines 11-14: **implementation of algorithm**
- line 16: **output**

Lexical components of a Java program

The source code of a Java program includes:

- names (of variables, methods, classes, ...)
- literals (constants)
- operators
- separation characters
- reserved terms
- comments
- white space

Reserved terms in Java

The following lists contains the Java keywords with a special meaning:

<code>abstract</code>	<code>assert</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>
<code>case</code>	<code>catch</code>	<code>char</code>	<code>class</code>	<code>continue</code>
<code>default</code>	<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>
<code>extends</code>	<code>final</code>	<code>finally</code>	<code>float</code>	<code>for</code>
<code>if</code>	<code>implements</code>	<code>import</code>	<code>instanceof</code>	<code>int</code>
<code>interface</code>	<code>long</code>	<code>native</code>	<code>new</code>	<code>package</code>
<code>private</code>	<code>protected</code>	<code>public</code>	<code>return</code>	<code>short</code>
<code>static</code>	<code>strictfp</code>	<code>super</code>	<code>switch</code>	<code>synchronized</code>
<code>this</code>	<code>throw</code>	<code>throws</code>	<code>transient</code>	<code>try</code>
<code>void</code>	<code>volatile</code>	<code>while</code>		

Reserved but not in use:

<code>const</code>	<code>goto</code>
--------------------	-------------------

Reserved terms are 'terminals' of the language definition and must not be used as names for variables etc.!

Comments

- 'Comments' are used to facilitate reading the source code of a program.
- Comments do not influence program execution!
- Without comments, programs often are so hard to understand that even the author does not understand them after a short while
- It cannot be expected from a corrector to grade programs without comments! From now on: programs without or with not enough comments will receive fewer points!
- Possible forms of comments in Java:

```
1  /*      a comment  */  
2  //      a comment until the end of the line  
3  /**     documentation that can be processed by javadoc  */
```

White space

- White space makes a program more readable.
- Java provides the following options for inserting white space:
 - ▶ **SP** (*space* with the space key)
 - ▶ **HT** (*horizontal tab* with the tab key)
 - ▶ **FF** (*form feed*)
 - ▶ **LF** (*line feed* with the enter key)
 - ▶ **CR** (*carriage return*)
- While space between symbols is ignored.
- White space within symbols is a (syntax) error
- For a guideline how comments and white space should be used see '*Code Conventions for the Java Programming Language*'
(<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>)

Important for comments and white space:

- Software is maintained during 80% of its life time.
- This maintenance is hardly ever done only by the original author!
- Following the conventions improves readability of software, such that it can be understood more quickly and more easily.

Separation characters

Separation characters delimit instructions, conditions, blocks, etc.

- '(' and ')' are the 'standard parenthesis', especially in 'expressions'.
- '[' and ']' are used with arrays.
- '{' and '}' mark blocks in programs, which can be used to merge multiple instructions to one (composite) instruction.
- ';' finishes instructions
- ',' is used for lists of things

Operators

Operators are used to write down arithmetic operations, arithmetic comparisons, logical connectives, and bit operations:

=	<	>	!	~	?	:				
==	<=	>=	!=	&&		++	--			
+	-	*	/	&		^	%	<<	>>	>>>
+=	-=	*=	/=	&=	=	^=	%=	<<=	>>=	>>>=

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- **Variable – Name – Value – Type**
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

To store data in programs **variables** are used:

- Variables had originally been abstractions of memory cells.
- They have a **name**, a **type**, and a **value**.
- They have to be declared before they can be used.
- They change their value due to assignments, increments, or decrements.
- They always contain a value which is compatible to their type.
- *Local variables* are valid from their declaration to the end of the corresponding block. (see later)
- *Local variables* must be explicitly assigned a value before they can be read. (see later)

A **variable name** consists of a sequence of uppercase and lowercase letters, digits, and some special characters.

- Be careful: Java distinguishes uppercase and lowercase letters!
- `'myVar'`, `'myvar'`, `'myVAR'` are different names!
- Names should follow the rules in the *Code Conventions for the Java Programming Language* .
- Thus all variable names should begin with a lowercase letter ...
- ... and in compound names the inner names should begin with a capital letter ('camelCase')
- good names:
`myVariable`, `i`, `length`, `averageCircleSize`, `newPrize1`, ...
- bad names:
`Myvariable`, `LENGTH`, `_i`, `$var`, `averagecirclesize`,...

Every variable has a '**type**' that determines which type of data it can store.

Examples for types in Java are:

- **byte** (for integers from **-128** to **127**)
- **short** (for integers from **-32768** to **32767**)
- **int** (for integers from **-2147483648** to **2147483647**)
- **long** (for larger integers until about $\pm 9 \cdot 10^{18}$)
- **float** (for floating point numbers, pretty inexact)
- **double** (for floating point numbers, less inexact)
- **char** (for single letters)
- **String** (for texts)

(more details later...)

```
1 int a, b;           // a and b are declared with type 'int'
2 int c = 1, d = 3;   // declaration with initialization
3 b = 1;              // assignment of value 1 to b
4 a = b + 2;          // assignment of value 3 to a
```

Examples for usage of variables

```
1 int width, length;
2 float myReal, yourFloat;
3
4 int i =1, j, k = 0;           // declaration with initialization
5
6 short _short, sHort, $short; // possible, but bad style
7
8 { int x = 1, y;               // declaration at start of block, if possible
9     y = x; }
10
11 { int x = 0;
12     x = x + 1; }             // an assignment is not an equation !!
13
14 int x = 3, y = 5;
15 { int help;                   // swap values
16     help = x; x = y; y = help; } // of x and y
```

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- **Literals and Constants**
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

Literals are used to represent numbers in the **source code of the program**

(in contrast to their internal representation in the computer)

- As **integers** are interpreted:
 - ▶ sequences of digits
 - ▶ sequences of digits that end with **1** or **L**
 - ▶ leading **0x** followed by hexadecimal digits, maybe with trailing **1/L**
- Literals that are interpreted as integers are mapped to the type **int** or the type **long** (requires suffix '**l**' or '**L**')

793677, 1, 35	int
0L, 17L, 30941	long
0xDadaCafe	int , hexadecimal (prefix '0x')
0xC0B0L	long , hexadecimal (prefix '0x', suffix 'L')
01234567	int , attention: octal (prefix '0')
- 23	not a literal (but: operator and literal)
9876543210	compiler error, value too large for int

- Sequences of digits that include at least one of the following characters are interpreted as **floating point numbers**
 - '.' as decimal point
 - E** or **e** for the exponent (decimal, scientific notation)
 - suffix 'F' or 'f' (to explicitly select the type **float**)
 - suffix 'D' or 'd' (to explicitly select the type **double**)

- Examples are:

75.286	double
.17E-9, 3.e+15, 5E19	double
14d, 4711D	double
.17E-9F, 3.e+15f	float
9876543210f	float
1e50f	compiler error, value too large for float

- Tip: use only the type **double**...!

- A variable that is declared with the prefix '**final**' is an (immutable) '**constant**':

```
1 final float PI = 3.14f;  
2 final int ONE = 1, NULL = 0;  
3 final int MY_INTEGER, YOUR_INTEGER;
```

- Predefined are for example **Double.NaN**, **Double.MAX_VALUE**, **Integer.MIN_VALUE** or **Long.MAX_VALUE**
- Names should follow the rules in the '*Code Conventions for the Java Programming Language*', for example
 - ▶ the name of constants should consist only of capitals.
 - ▶ the names of compound names should be connected with '*underscore*' ('_').

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- **Truth values**
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

boolean is the data type for truth values (after George Boole).

Possible values are **true**, **false**

Examples:

```
1 boolean bool1, bool2, bool3;
2 bool1 = true;
3 int i = 1;
4 bool2 = i == 0;    // yields 'false'
5 if (true) bool3 = false;
6 if (bool2) System.out.print("i_is 0");
```

- Operators to generate truth values:
'==', '!=', '<', '<=', '>', '>='
- Operators to combine truth values:
'!', '&&', '||'
- more details later...

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- **Assignments and Expressions**
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

- *Syntax* of **assignments**:

x = expression

If **x** has one of the types **byte**, **short**, **int**, **long**, **float**, **double**,

then **expression** resembles arithmetic expressions from elementary mathematics

and consists of variables, constants, operators, and parentheses.

- *Semantics* of assignments:

Compute the value of **expression** and assign it to the variable **x**.

- ▶ Variables appearing in **expression** must have a value.
- ▶ The type of **expression** must be compatible to the type of **x**.

- In addition the value of an assignment is the assigned value, i.e., '=' is also an operator!

Example: '**x=y=2**'

Arithmetic Operators

- Operators for addition '+' and subtraction '-'
 - ▶ Example: $x = -(y + z);$
 - ▶ Example: $x = +y + (z - w);$
- Operator for multiplication '*'
 - ▶ Example: $x = y * (z + 7)$
- Operator for division '/'
 - ▶ Example: $x = z / y$
 - ▶ Important: With integer values the result is rounded towards zero, i.e., $99/100$ yields 0, similar to $(-99)/100$

- Operator for remainder with Euclidean division ‘%’
 - ▶ Example: $x = z \% y$
 - ▶ If z and y are both non-negative, then this is the modulo function, $9\%5$ yields 4

Be careful with negative arguments:

- ▶ in Java/C/JavaScript/PHP $z\%y$ is equivalent to $z - (z/y) * y$, which is *not* the modulo function for negative arguments.
 $(-9)\%5$ thus yields -4
 - ▶ In Perl/Python/Ruby $z\%y$ is equivalent to $z - \lfloor \frac{z}{y} \rfloor * y$.
 $(-9)\%5$ yields 1 in this case
- Operators for increment ‘++’ and decrement ‘--’
 - ▶ Example: $x++$ corresponds to $x = x + 1$
 - ▶ Example: $x--$ corresponds to $x = x - 1$
- ... and more (see later)

Operator precedence:

In expressions that are not fully parenthesized the order of execution is determined by operator precedence (or order of operations):

Example: $a - b * c / d + e$ is interpreted as $((a - (b * c) / d) + e)$

The predefined precedence rules are:

precedence-level	operator	operation	processing-direction
1	'++'	increment (postfix)	L->R
	'--'	decrement (postfix)	
2	'+'	plus (unary, prefix)	R->L
	'-'	minus (unary, prefix)	
3	'*'	multiplication	L->R
	'/'	division	
	'%'	remainder	
4	'+'	addition	L->R
	'-'	subtraction	
5	'='	assignment	R->L

Type conversion with elementary types:

- It is possible to mix different types in arithmetic expressions,
- this requires conversion between the types.

In Java conversions are used in three ways:

- with an assignment
- during the evaluation of arithmetic expressions
- with explicit casting

- **'widening'** = conversion into a type with larger value space, therefore (usually) unproblematic

'Widening' done automatically when required:

byte	in	short,	int,	long,	float,	double
short	in		int,	long,	float,	double
char	in		int,	long,	float,	double
int	in			long,	float⁽¹⁾,	double
long	in				float⁽¹⁾,	double⁽¹⁾
float	in					double

With ⁽¹⁾ some precision may be lost, see below

- **'narrowing'** = conversion into type with smaller/inappropriate value space.
 - ▶ here we may lose information,
 - ▶ since the target type may not be able to represent the original value.
 - ▶ this requires an explicit type conversion ('cast').
 - ▶ **double → float → long → int → short/char → byte**

Example: consider variables `float floatNumber; int intNumber;`

- conversion during assignment (only 'widening' allowed):
 - ▶ `floatNumber = intNumber;`
- Conversion during the evaluation of arithmetic expressions:
 - ▶ `floatNumber = floatNumber / intNumber;`
division with `float`-copy of `intNumber` (widening!)
- conversion with explicit casting, in particular with narrowing:
 - ▶ `intNumber = (int) floatNumber;`

Examples for casting:

(float)	1.79E+308	→	infinity
(long)	1.79E+308	→	9223372036854775807
(int)	1.79E+308	→	2147483647
(short)	2147483647	→	-1
(int)	1.0e6	→	1000000
(int)	1.11	→	1
(int)	-1.49	→	-1
(byte)	259	→	3

Conditions are required to generate Boolean values
(for example for branches in the program execution)

- The evaluation of a condition yields either **true** or **false**.
- Possible forms of conditions:
 - ▶ **true**
 - ▶ **false**
 - ▶ simple condition:

expression comparison_operator expression

- ▶ compound condition:

unary_logical_operator condition

or

condition binary_logical_operator condition

Comparison operators yield values of the type **boolean**

Java operator	mathematical symbol	pronounced
<code>==</code>	$=$	'equal to'
<code>!=</code>	\neq	'not equal to'
<code><</code>	$<$	'less than'
<code><=</code>	\leq	'less than or equal to'
<code>></code>	$>$	'greater than'
<code>>=</code>	\geq	'greater than or equal to'

Logical operators combine values of the type `boolean`

Java operator	mathem. symbol	pronounced	precedence	order
unary, i.e., one operand				
!	\neg	'not'	1	$R \rightarrow L$
binary, i.e., two operands				
&&	\wedge	'and'	2	$L \rightarrow R$
 	\vee	'or'	3	$L \rightarrow R$

Truth tables:

A	!A
true	false
false	true

A	B	A&&B	A B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- **Basic control structures in Java**
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

Blocks / Sequences

Constructing **blocks** from single instructions is one of the basic operations to structure programs!

- With '{' and '}' it is possible to combine multiple instructions to a block (in the sense of a **sequence**).
- A block can be used wherever an instruction can be used.
- The boundaries of a block restrict the life span of variables.
- '{ }' represents the empty block.
- Instead of '{ }' one can also use ';' for an empty instruction.

```

1 //correct block construction!
2 public class K3B02E_Block {
3     public static void
4         main(String[] args) {
5
6         int i = 3;
7         if (i < 0) {}
8         else {
9             String out;
10            out = "i_:_" + i;
11            System.out.println(out);
12        }
13    }
14 }

```

```
>javac K3B02E_Block.java
```

```
>java K3B02E_Block
i : 3
```

```

1 //wrong block construction!
2 public class K3B03E_Block {
3     public static void
4         main(String[] args) {
5
6         int i = 3;
7         if (i < 0) {}
8         else {
9             String out;
10            out = "i_:_" + i;
11        }
12            System.out.println(out);
13    }
14 }

```

```

>javac K3B03E_Block.java
K3B03E_Block.java:11: error: cannot find symbol
    System.out.println(out);
                        ^

```

```

    symbol:   variable out
    location: class K3B03E_Block
1 error

```

Branching instructions with `if`

possible instances:

- `if (condition) sequence1 else sequence2`
- `if (condition) sequence`

```
1 public class K3B04E_Minimum {
2     public static void main(String[] args) {
3         int num1, num2, num3, min;
4         num1 = Integer.parseInt(args[0]);
5         num2 = Integer.parseInt(args[1]);
6         num3 = Integer.parseInt(args[2]);
7
8         if (num1 < num2)
9             if (num1 < num3) min = num1;
10            else min = num3;
11        else
12            if (num2 < num3) min = num2;
13            else min = num3;
14
15        System.out.println("Minimum:_" + min);
16    }
17 }
```



```

1 import javax.swing.JOptionPane;
2 public class K3B05E_Branches {
3     public static void main(String[] args) {
4
5         int points = 0, grade = 0;
6         // list of declarations with initialization
7         String inWords = "wrong input";
8         // more on Strings later...
9         points = Integer.parseInt(
10             JOptionPane.showInputDialog ("Points: ") );
11
12         if (points < 40 && points >= 0)
13             { grade = 5; inWords = "not sufficient"; }
14         if (points >= 40 && points < 50 )
15             { grade = 4; inWords = "sufficient"; }
16         if (points >= 50 && points < 60 )
17             { grade = 3; inWords = "satisfactory"; }
18         if (points >= 60 && points < 70 )
19             { grade = 2; inWords = "good"; }
20         if (points >= 70 && points <= 80)
21             { grade = 1; inWords = "very good"; }
22
23         JOptionPane.showMessageDialog (null,
24             "grade: " + grade + " (" + inWords + ")");
25     }
26 }

```

```
1 import javax.swing.JOptionPane;
2 public class K3B06E_nested_Branches {
3     public static void main(String[] args) {
4
5         int points = 0, grade = 0;
6         String inWords = "wrong input";
7         points = Integer.parseInt(
8             JOptionPane.showInputDialog ("points: "));
9
10        if (points < 0  || points > 80) {}
11        else
12            if (points < 40) { grade = 5; inWords = "not sufficient"; }
13            else
14                if (points < 50 ) { grade = 4; inWords = "sufficient"; }
15                else
16                    if (points < 60 ) { grade = 3; inWords = "satisfactory"; }
17                    else
18                        if (points < 70 ) { grade = 2; inWords = "good"; }
19                        else
20                            { grade = 1; inWords = "very good"; }
21
22        JOptionPane.showMessageDialog (null,
23            "grade: " + grade + " (" + inWords + ")");
24    }
25 }
```

Be careful with **dangling else**:

- Every **if** can come with an optional(!) **else**...
- Problem with two nested **ifs** with only one **else**:

```
1 if (a == 1)
2     if (b == 1)
3         c = 42;
4 else
5     d = 42;
```

- To which **if** does the **else** belong? When are **c** or **d** changed?
- Basic rule in **JAVA**, **C**, **C++**:
 else belongs to the directly preceding **if**.
- In the example: If **a** is not **1**, then neither **c** nor **d** are changed.
 If **a=1** and **b≠1**, then only **d** is changed.
- Better: mark all alternatives with { } as blocks.
- other programming languages (**ADA**, **Delphi**, **BASIC**,...):
 usually use bracketing with closing **endif**, **fi** or similar

The previous example is thus equivalent to

```
1 if (a == 1) {  
2     if (b == 1)  
3         c = 42;  
4     else  
5         d = 42;  
6 }
```

Better: always use brackets, e.g.:

```
1 if (a == 1) {  
2     if (b == 1) {  
3         c = 42;  
4     } else {  
5         d = 42;  
6     }  
7 }
```

Branching instruction with **switch**:

With a larger number of alternatives nested **if** blocks quickly get confusing, then it is better to use **switch**:

- **switch** with **break** (causes remaining cases to be skipped...)

```
1 switch (expression){  
2     case value1: sequence1; break;  
3     case value2: sequence2; break;  
4     ...  
5     case valueN: sequenceN; break;  
6 [ default: defaultSequence ] // optional  
7 }
```

- **switch** without **break**

```
1 switch (expression){  
2     case value1: sequence1;  
3     case value2: sequence2;  
4     ...  
5     case valueN: sequenceN;  
6 [ default: defaultSequence ] // optional  
7 }
```

```
1 import javax.swing.JOptionPane;
2 public class K3B07E_Switch {
3     public static void main(String[] args) {
4
5         int points = 0, case, grade = 0;
6         String inWords = "wrong input";
7         points = Integer.parseInt(
8             JOptionPane.showInputDialog ("points: "));
9
10        if (points < 0 || points > 80) {}
11        else {
12            case = points / 10;
13            switch (case) {
14                case 8:
15                    case 7: grade = 1; inWords = "very good"; break;
16                    case 6: grade = 2; inWords = "good"; break;
17                    case 5: grade = 3; inWords = "satisfactory"; break;
18                    case 4: grade = 4; inWords = "sufficient"; break;
19                    default: grade = 5; inWords = "not sufficient";
20            }
21        }
22
23        JOptionPane.showMessageDialog (null,
24            "grade: " + grade + "_" + inWords + "");
25    }
26 }
```

```

1 public class K3B08E_Switch_without_Break {
2     public static void main(String[] args) {
3         int in = 0, out = 0;
4         in = Integer.parseInt( args[0] );
5
6         switch (in) {
7             case 8:
8             case 7: out = out + 1 ;
9             case 6: out = out + 1 ;
10            case 5: out = out + 1 ;
11            case 4: out = out + 1 ;
12            case 3:
13            case 2:
14            case 1:
15            case 0: out = out + 1 ;
16        }
17
18        System.out.println ( "Input: " + in);
19        System.out.println ( "Output: " + out);
20    }
21 }

```

Input:		0	1	2	3	4	5	6	7	8
Output:		1	1	1	1	2	3	4	5	5

Java essentially provides three kinds of **iterations / loops**: **while**, **do-while**, and **for**.

- Syntax of the **while** loop

```
1 while (condition) sequence
```

- Semantics:

- ▶ Evaluate the value of **condition**.
- ▶ If value is **false**, stop execution of the loop.
- ▶ If value is **true**, execute **sequence**.
- ▶ repeat the loop...

- sequence of processing:

- ▶ **condition true**
- ▶ **sequence**
- ▶ **condition true**
- ▶ **sequence**
- ▶ ...
- ▶ **condition false**

- If **condition** is already **false** in the beginning, **sequence** is not executed at all.

Example: Given input n , compute $\sum_{i=1}^n i$ with **while** loop.

```
1 public class K3B09E_While {
2     public static void main(String[] args) {
3
4         int i = 1, n = 0, sum = 0;
5         n = Integer.parseInt( args[0]);
6
7         while (i <= n) {    // correct for n >= 0
8             sum = sum + i;
9             i++;
10        }
11
12        System.out.println( "For n = " + n
13                            + "_the sum is = " + sum);
14    }
15 }
```

- Syntax of the **do-while** loop:

```
1  do sequence while (condition)
```

- Semantics:

- ▶ execute **sequence**.
- ▶ evaluate the value of **condition**.
- ▶ If value is **false**, stop execution of the loop.
- ▶ If value is **true**, repeat the loop...

- sequence of processing:

- ▶ **sequence**
- ▶ **condition true**
- ▶ **sequence**
- ▶ ...
- ▶ **condition false**

- **sequence** is always executed at least once!
- **sequence** is often called the body of the loop.
- If **condition** never gets false: 'infinite loop'

```
1 public class K3B10E_DoWhile {
2     public static void main(String[] args) {
3
4         int i = 1, n = 0, sum = 0;
5         n = Integer.parseInt( args[0]);
6
7         do { // correct only for n > 0
8             sum = sum + i;
9             i++;
10        } while (i <= n)
11
12        System.out.println( "For n = " + n
13            + "_the sum is = " + sum);
14    }
15 }
```

Example for complex loops: “Collatz Problem”,
“(3n+1) Conjecture”

Consider number sequences constructed using the following rules:

- Start with an arbitrary natural number $n > 0$.
- If n is even, then continue with $n/2$.
- If n is odd, then continue with $3 \cdot n + 1$.

Examples:

- $12 \rightsquigarrow 6 \rightsquigarrow 3 \rightsquigarrow 10 \rightsquigarrow 5 \rightsquigarrow 16 \rightsquigarrow 8$
 $\rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \dots$
- $14 \rightsquigarrow 7 \rightsquigarrow 22 \rightsquigarrow 11 \rightsquigarrow 34 \rightsquigarrow 17 \rightsquigarrow 52 \rightsquigarrow 26 \rightsquigarrow 13$
 $\rightsquigarrow 40 \rightsquigarrow 20 \rightsquigarrow 10 \dots$

Thus: we will often (or always?) eventually get the sequence
 $1 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \dots$

Implementation with nested loops:

```
1 public class K3B11E_Collatz {
2     public static void main(String[] args){
3
4         int n = 0, inner = 0, outer = 0;
5         n = Integer.parseInt( args[0] );
6
7         while (n > 1) {
8             if (n % 2 != 0) {
9                 n = 3 * n + 1; outer ++;
10            }
11            while (n % 2 == 0) {
12                n = n / 2; inner ++;
13            }
14        }
15        System.out.println( "n=_l_after " + inner
16                            + "_steps in the inner loop and " + outer
17                            + "_steps in the outer loop");
18    }
19 }
```

- How many iterations inner/outer are there for an input?

Still unsolved Collatz conjecture:

- does the problem stop at all for every input?

(introduced in 1937 by Lothar Collatz)

<i>n</i>	inner	outer
11	10	4
101	18	7
1001	91	51
10001	115	64

- Syntax of the **for** loop:

```
1 for ( initialization(s); condition; assignment(s) )  
2     sequence
```

- Semantics:

- ▶ First the initializations are executed.
- ▶ Then **condition** is evaluated.
- ▶ If the condition is satisfied, **sequence** is executed.
- ▶ Then the assignments are executed and the execution restarts with the condition test.

- Processed like the following while loop:

```
1 initialization(s);  
2 while (condition) {  
3     sequence;  
4     assignment(s)  
5 }
```

```
1 public class K3B12E_For {
2     public static void main(String[] args) {
3
4         int i = 1, n = 0, sum = 0;
5         n = Integer.parseInt( args[0]);
6
7         for (i = 1; i <= n; i++) {
8             sum = sum + i;
9         }
10
11         System.out.println( "For n = " + n
12             + " the sum is = " + sum);
13     }
14 }
```


for loops can be very complex:

```
1 public class K3B13E_Collatz_with_For {
2     public static void main(String[] args) {
3
4         int n = 0, inner = 0, outer = 0;
5         n = Integer.parseInt( args[0] );
6
7         for ( ;
8             n > 1 ;
9             n= (n%2!=0) ? (3 * n + 1 + 0*(outer++))
10                : (n / 2      + 0*(inner++ )) )
11         {}
12
13         System.out.println( "n=_l_after " + inner
14                             + "_steps in the inner loop and " + outer
15                             + "_steps in the outer loop");
16     }
17 }
```

- Here, the iteration of the Collatz problem is done solely in the loop parameters.
- Almost unreadable program, very bad programming style!
- Please do not imitate!

Usual application of for loops: **nested counting loops**

```
1 public class K3B14E_Flag {
2     public static void main(String[] args) {
3         int i, j, flagSize;
4         flagSize = Integer.parseInt(
5             System.console().readLine("size of the flag: "));
6         for (i = 1; i <= flagSize; i++) {
7             System.out.print("|");
8             for (j = 1; j <= flagSize; j++)
9                 if (i == j || (flagSize + 1 - j) == i)
10                     System.out.print("xxx");
11                 else
12                     System.out.print("_");
13             System.out.println("|");
14         }
15     }
16 }
```

output for **flagSize 7**:

```
|xxx          xxx|
|   xxx      xxx |
|       xxx  xxx  |
|         xxx     |
|      xxx  xxx   |
|   xxx      xxx  |
|xxx          xxx|
```

Comparison of `break` and `continue` :

- If the instruction

`continue;`

is executed within a loop

(usually in an if instruction...),

then *the current iteration is stopped* and the loop continues with the next iteration.

- If the instruction

`break;`

is executed within a loop

(usually in an if instruction...),

then *the whole loop is stopped* and the execution continues with the first instruction after the loop.

(analogously to `break` with `switch...`)

```

1 public class K3B15E_Break {
2     public static void
3         main(String[] args) {
4
5         System.out.println("Start");
6         for (int i = 0; i <= 6; i++) {
7             if (i == 3) break;
8             System.out.println(i);
9         }
10
11        System.out.println("End");
12    }
13 }

```

>java K3B15E_Break

Start

0

1

2

End

>

>

>

```

1 public class K3B16E_Continue {
2     public static void
3         main(String[] args) {
4
5         System.out.println("Start");
6         for (int i = 0; i <= 6; i++) {
7             if (i == 3) continue;
8             System.out.println(i);
9         }
10
11        System.out.println("End");
12    }
13 }

```

>java K3B16E_Continue

Start

0

1

2

4

5

6

End

>

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- **Integer Numbers**
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

From Bits to Numbers

- A computer represents all data in binary form (using transistors, i.e., electrical switches, values: on/off)
- Interpretation: switch on = **1**, switch off = **0**
- Instead of single bits, groups of bits are read and set
 - ▶ 1 bit = 1 'switch', 1 byte = 8 bits
- The main memory of a computer is a list of bytes, e.g.,:

address	...	40	41	42	43	...
content	...	01100111	00010010	00000000	00000000	...

- Recent CPUs usually process 32 bit or 64 bit at once.
- Memory accesses thus always start addresses divisible by 4 (or 8)
- In the example: 32bit-access to address 40 yields the bit group

00000000 00000000 00010010 01100111

(usually shown with descending addresses ('little endian'))

To ease readability:

- groups of 4 bits are often represented by a single character ('nibble', 'half byte') and denoted 'hexadecimally'



nibble	0000	0001	0010	0011	0100	0101	0110	0111
hexadecimal	0	1	2	3	4	5	6	7
decimal value	0	1	2	3	4	5	6	7

nibble	1000	1001	1010	1011	1100	1101	1110	1111
hexadecimal	8	9	A	B	C	D	E	F
decimal value	8	9	10	11	12	13	14	15

- Example: 00000000 00000000 00010010 01100111
hexadecimal representation **00 00 12 67**
- To indicate hexadecimal representation, Java uses a leading **0x**, e.g., '**a=0x00001267**'
- Less frequently used: groups of 3 bits, 'octal' representation, in Java indicated by leading 0,
e.g.,: '**a=0715**' for the octal number '**111 001 101**'
(decimal value: $7 \cdot 8^2 + 1 \cdot 8 + 5 = 461$)

Binary representation of natural numbers:

- Numbers are usually represented with 32 or 64 bits:

Example: the bit pattern **0x00001267**, binary:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 0 1 1 1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

can be interpreted as a (binary) number as follows :

$$\begin{aligned} & 1 \cdot 2^{12} + 1 \cdot 2^9 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ = & 4096 + 512 + 64 + 32 + 4 + 2 + 1 \\ = & 4711 \end{aligned}$$

- alternatively: evaluate **0x00001267** directly (with base 16):

$$\begin{aligned} & 1 \cdot 16^3 + 2 \cdot 16^2 + 6 \cdot 16^1 + 7 \cdot 16^0 \\ = & 4096 + 512 + 16 + 7 = 4711 \end{aligned}$$

Binary representation of natural numbers/integers:

- The k -bit pattern $b_{k-1} \dots b_2 b_1 b_0$ represents the **natural number**

$$\sum_{i=0}^{k-1} b_i \cdot 2^i$$

- Usually, however, the ‘**two’s complement**’ representation is used, where $b_{k-1} \dots b_2 b_1 b_0$ corresponds to the **integer**

$$\sum_{i=0}^{k-2} b_i \cdot 2^i - b_{k-1} \cdot 2^{k-1}$$

- Examples for $k = 8$, i.e., interpretation of a byte as an integer:

byte	computation	value
0 000 0000	0 – 0	0
0 000 0001	1 – 0	1
0 000 0101	5 – 0	5
0 101 0101	85 – 0	85
0 111 1111	127 – 0	127

byte	computation	value
1 000 0000	0 – 128	–128
1 000 0001	1 – 128	–127
1 000 0101	5 – 128	–123
1 101 0101	85 – 128	–43
1 111 1111	127 – 128	–1

Primitive Java data types for integers

With k bits one can represent

- negative integers from -2^{k-1} to -1 ,
- positive integers from 1 to $2^{k-1}-1$, (asymmetric!)
- and the 0 .

Java uses two's complement for integers, with the following number ranges:

Java type			minimal value	maximal value
byte	1 byte	8 bit	-128	127
short	2 bytes	16 bits	-32768	32767
int	4 bytes	32 bits	-2147483648	2147483647
long	8 bytes	64 bits	-9223372036854775808	9223372036854775807

Remark:
$$\sum_{i=0}^{k-2} b_i \cdot 2^i - b_{k-1} \cdot 2^{k-1} = \sum_{i=0}^{k-1} b_i \cdot 2^i - b_{k-1} \cdot 2^k$$

Therefore a computer can use the same(!) hardware for addition/subtraction/multiplication of integers and for natural numbers!

- Attention: the computer uses only the last bits of a number, all other bits are **discarded without warning!**
- This corresponds to computing modulo 2^k (with sign according to two's complement)
- Example: addition of the decimal values **1 000 000 000** and **2 000 000 000** with `int`, i.e., $k = 32$:

$$\begin{array}{r}
 00111011\ 10011010\ 11001010\ 00000000 \\
 +\ 01110111\ 00110101\ 10010100\ 00000000 \\
 =\ 10110010\ 11010000\ 01011110\ 00000000
 \end{array}$$

The result is **negative** (two's complement!) with value

$$3\ 000\ 000\ 000 - 2^{32} = -1\ 294\ 967\ 296$$

- The programmer must take care that no such '**overflow**' happens!

Conversion rules for integers in Java:

- Java has integer types **byte**, **short**, **int**, and **long**
- Numbers are always stored in two's complement!
- **byte** and **short** numbers are always converted to **int** before any computation!
- However, results are not automatically converted to **byte** and **short**!

```
1    byte b1,b2;
2    b1=101;  b2=102;
3
4    int i;
5    i = b1 + b2;
6    System.out.println("as int: " + i);    // result: 203
7
8    byte b;
9    // b = b1 + b2;           // wrong! -> compiler indicates error
10   b = (byte) (b1 + b2);    // ok with 'cast' (see later)
11   System.out.println("as byte: " + b);    // result: -53
```

Identifying and dealing with integer overflows is a task of the programmer!

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- **Floating Point Numbers**
- Characters and Strings
- Additional Operators
- Methods

property	32 bit / float	64 bit / double
greatest positive (finite) number	$2^{128} - 2^{104} \approx 2^{128}$ $\approx 3.4 \cdot 10^{38}$	$2^{1024} - 2^{971} \approx 2^{1024}$ $\approx 1.8 \cdot 10^{308}$
smallest possible normalized number	2^{-126} $\approx 1.2 \cdot 10^{-38}$	2^{-1022} $\approx 2.2 \cdot 10^{-308}$
smallest possible denormalised number	2^{-150} $\approx 7 \cdot 10^{-46}$	2^{-1075} $\approx 4.94 \cdot 10^{-324}$

Floating point numbers as primitive data types:

```
1 public class K3B17E_double_float {
2     public static void main(String[] args) {
3         float xf;
4         double xd;
5         xf = 123456.78901234567890f;
6         xd = 0.012345678901234567890d;
7         System.out.println( xf );
8         System.out.println( xd );
9         System.out.println( 0.12345e-5 );
10        System.out.println( 1.0e200 * 1.0e200 );
11        System.out.println( 1.0 / 0.0 );
12        System.out.println( -1.0 / 0.0 );
13        System.out.println( 0.0 / 0.0 );
14    }
15 }
```

```
1 123456.79
2 0.012345678901234568
3 1.2345E-6
4 Infinity
5 Infinity
6 -Infinity
7 NaN
```

Comparing **float** (or **double**) numbers is dangerous:

```
1 public class K3B18E_inexact {
2     public static void main(String[] args) {
3
4         float xf = 1.0f, yf = 1.0f;
5         xf = xf / 3 * 100000 * 3 / 100000; // value = 1 , or...?
6         System.out.print ( "xf:_ " + xf + "\nyf:_ " + yf + "\n");
7         if (xf == yf) System.out.println ("xf_and yf are equal ");
8         else          System.out.println ("xf_and yf are not equal");
9
10        double xd = 1.0, yd = 1.0;
11        xd = xd / 3 * 100000 * 3 / 100000; // value = 1 , or...?
12        System.out.print ( "xd:_ " + xd + "\nyd:_ " + yf + "\n");
13        if (xd == yd) System.out.println ("xd_and yd are equal");
14        else          System.out.println ("xd_and yd are not equal");
15    }
16 }
```

```
1 xf: 1.0000001
2 yf: 1.0
3 xf and yf are not equal
4 xd: 0.9999999999999999
5 yd: 1.0
6 xd and yd are not equal
```


'Tolerant' comparison of **float** numbers (or **double** numbers)

```
1 public class K3B19E_Tolerance {
2     public static void main(String[] args) {
3
4         double x = 1.0, y = 1.0, tolerance = 0.001;
5
6         x = x / 3 * 100000 * 3 / 100000;
7
8         System.out.print ("x:_ " + x + "\ny:_ " + y + "\n");
9         if (Math.abs (x - y) < tolerance) // absolute value of (x-y)
10             System.out.println("x_and y are almost equal");
11         else
12             System.out.println("x_and y are probably not equal");
13     }
14 }
```

```
1 x: 0.9999999999999999
2 y: 1.0
3 x and y are almost equal
```

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- **Characters and Strings**
- Additional Operators
- Methods

- Purpose of the type **char**: storing single characters
- character set: Unicode,
represented as groups of 2 bytes
- characters can be given in hexadecimal form `\u0000` to `\uFFFF`
(anywhere in the source code!)
- ‘readable’ character literals with apostroph (‘ ’)

in source code	meaning
<code>char character = 'a'</code>	<code>\u0061</code> , ASCII, printable
<code>char percent = '%'</code>	<code>\u0025</code> , ASCII, printable
<code>char newline = '\n'</code>	<code>\u000A</code> , ASCII ‘control character’
<code>char c_return = '\r'</code>	<code>\u000D</code> , ASCII ‘control character’
<code>char backslash = '\\'</code>	<code>\u005C</code> , ASCII, printable
<code>char quote = '\"'</code>	<code>\u0060</code> , ASCII, printable
<code>char omega = '\u03a9'</code>	Ω , Unicode hexadecimal
<code>char sigma = '\Sigma'</code>	Σ , Unicode direct

- terminology:
`\n`, `\r`, `\\`, `\'` etc. are called ‘escape sequences’
`\u03a9` etc. are called ‘Unicode escape sequences’

- first 128 characters `\u0000` to `\u007F` in the Unicode charset:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	<code>nul</code>	<code>soh</code>	<code>stx</code>	<code>etx</code>	<code>eot</code>	<code>enq</code>	<code>ack</code>	<code>bel</code>	<code>bs</code>	<code>ht</code>	<code>nl</code>	<code>vt</code>	<code>ff</code>	<code>cr</code>	<code>so</code>	<code>si</code>
1	<code>dle</code>	<code>dc1</code>	<code>dc2</code>	<code>dc3</code>	<code>dc4</code>	<code>nak</code>	<code>syn</code>	<code>etb</code>	<code>can</code>	<code>em</code>	<code>sub</code>	<code>esc</code>	<code>fs</code>	<code>gs</code>	<code>rs</code>	<code>us</code>
2	<code>sp</code>	<code>!</code>	<code>"</code>	<code>#</code>	<code>\$</code>	<code>%</code>	<code>&</code>	<code>'</code>	<code>(</code>	<code>)</code>	<code>*</code>	<code>+</code>	<code>,</code>	<code>-</code>	<code>.</code>	<code>/</code>
3	<code>0</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>	<code>6</code>	<code>7</code>	<code>8</code>	<code>9</code>	<code>:</code>	<code>;</code>	<code><</code>	<code>=</code>	<code>></code>	<code>?</code>
4	<code>@</code>	<code>A</code>	<code>B</code>	<code>C</code>	<code>D</code>	<code>E</code>	<code>F</code>	<code>G</code>	<code>H</code>	<code>U</code>	<code>J</code>	<code>K</code>	<code>L</code>	<code>M</code>	<code>N</code>	<code>O</code>
5	<code>P</code>	<code>Q</code>	<code>R</code>	<code>S</code>	<code>T</code>	<code>U</code>	<code>V</code>	<code>W</code>	<code>X</code>	<code>Y</code>	<code>Z</code>	<code>[</code>	<code>\</code>	<code>]</code>	<code>^</code>	<code>_</code>
6	<code>`</code>	<code>a</code>	<code>b</code>	<code>c</code>	<code>d</code>	<code>e</code>	<code>f</code>	<code>g</code>	<code>h</code>	<code>i</code>	<code>j</code>	<code>k</code>	<code>l</code>	<code>m</code>	<code>n</code>	<code>o</code>
7	<code>p</code>	<code>q</code>	<code>r</code>	<code>s</code>	<code>t</code>	<code>u</code>	<code>v</code>	<code>w</code>	<code>x</code>	<code>y</code>	<code>z</code>	<code>{</code>	<code> </code>	<code>}</code>	<code>~</code>	<code>del</code>

- `\u0000` to `\u007F` are exactly the first 127 characters in the 7bit 'ASCII' charset (*American Standard Code for Information Interchange*, 1963)...
- ... and also in the 8bit 'ISO LATIN 1' charset (ISO-8859-1)

- Character string literals are assigned to the type **String**.
- **String** is not a primitive data type, but can partly be used like one (later more on the class **String**).

source code	meaning
"This is a string"	String with 16 characters
"this"+"string"	concatenated String
"\nString"	String with linefeed at the beginning
" "	empty String
"\""	String that contains only the character "

Example program:

```

























1 String chain = "This is a String";
2 int i = 27;
3 chain = chain + "_of length " + i;
4 /*      i is automatically converted          */
5 /*      to a String                          */
6 /*      value of chain: This is a String of length 27 */

```

To allow for more than $2^{16} = 65536$ different characters in a String:

- After a **char** from `\uD800 – \uDBFF` in a String only a **char** from `\uDC00 - \uDFFF` can follow
- These four bytes are always interpreted together!
- binary: values from `\uD800 – \uDBFF`: `1101 10xx xxxx xxxx`
- binary: values from `\uDC00 – \uDFFF`: `1101 11xx xxxx xxxx`
- The first 5 bits `1101 1` indicate if this is a part of a 'surrogate' pair!
- This yields 20 bits for up to **1048576** characters
- Example: Unicode character `U+1D160`
binary: `0001 1101 0001 0110 0000`
split in two groups of 10 bits: `00 0111 0100, 01 0110 0000`
surrogate pair: `\uD874 \uDD60`
- More information on Unicode at <http://www.unicode.org/>

Examples for Unicode tables:

							
1D100	1D110	1D120	1D130	1D140	1D150	1D160	1D170
							
1D101	1D111	1D121	1D131	1D141	1D151	1D161	1D171
							
1D102	1D112	1D122	1D132	1D142	1D152	1D162	1D172

ا	ز	ق	چ	ک	ن	ئ	ط		و	ئو	ئج	تي	صح	فم
FB50	FB60	FB70	FB80	FB90	FBA0	FB80	FBC0		FBE0	FBF0	FC00	FC10	FC20	FC30
ا	ز	ق	چ	ک	ن	ئ	ط		و	ئو	ئج	تي	صح	فم
FB51	FB61	FB71	FB81	FB91	FBA1	FB81	FBC1		FBE1	FBF1	FC01	FC11	FC21	FC31
پ	ت	ج	ي	گ	ڊ	ٻ			و	ئو	ئم	ثم	ضج	في
FB52	FB62	FB72	FB82	FB92	FBA2	FB82			FBE2	FBF2	FC02	FC12	FC22	FC32

<p>2F8E0 木 75.4</p> <p>枅 枅 T8-384A KP1-4B46</p> <p>≡ 6785 枅</p> <p>2F8E1 木 75.6</p> <p>𣏟 𣏟 T3-315C KP1-4B5D</p> <p>≡ 6852 𣏟</p> <p>2F8E2 木 75.7</p> <p>梅 梅 T4-2D5C KP1-4B57</p> <p>≡ 6885 梅</p>	<p>2F8EF 欠 76.2</p> <p>次 次 T6-2523</p> <p>≡ 6B21 次</p> <p>2F8F0 欠 76.6</p> <p>𣎵 𣎵 T6-4A3F</p> <p>≡ 238A7 𣎵</p> <p>2F8F1 欠 76.12</p> <p>𣎶 𣎶 T7-2378</p> <p>≡ 6B54 𣎶</p>	<p>2F8FE 水 85.4</p> <p>汧 汧 T3-2D52 KP1-511E</p> <p>≡ 6C67 汧</p> <p>2F8FF 水 85.7</p> <p>𣎵 𣎵 T6-3239</p> <p>≡ 6D16 𣎵</p> <p>2F900 水 85.6</p> <p>派 派 T6-3242 KP1-5145</p> <p>≡ 6D3E 派</p>
---	--	--

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- **Additional Operators**
- Methods

Increment and Decrement

expression	operation	value of the expression
count++	add 1	= value <i>before</i> the addition
++count	add 1	= value <i>after</i> the addition
count--	subtract 1	= value <i>before</i> the subtraction
--count	subtract 1	= value <i>after</i> the subtraction

```
1 int x = 0, count = 0;
2 x = count++;           // after instruction:  x: 0, count: 1
3 x = ++count;           // after instruction:  x: 2, count: 2
4 x = count--;           // after instruction:  x: 2, count: 1
5 x = --count;           // after instruction:  x: 0, count: 0
```

Examples:



```
x = i++
```

is equivalent to

```
x = i; i = i+1;
```



```
x = ++i
```

is equivalent to

```
i = i+1; x = i;
```



```
if (i ++ == 1) sequence
```

is equivalent to

```
if (i==1) {i=i+1; sequence} else {i= i+1;}
```

What does the following program?:

```
1 public class K3B20E_Increment {
2     public static void main(String[] args) {
3
4         int x = 0, count = 0;
5
6         ++count;
7
8         x = 5*(count++) + count--;
9         //allowed, but bad style...
10
11     System.out.println("x:_ " + x + " __count:_ "+count);
12 }
13 }
```

```
1 > java K3B20E_Inkrement
2 x: 7  count: 1
```

Assignment operators

operator	meaning	example	equivalent to
=	assignment	x=y	
+=	addition, then assignment	x+=y	x=x+y
+=	concatenation, then assignment	x+=y	x=x+y
-=	subtraction, then assignment	x-=y	x=x-y
=	multiplication, then assignment	x=y	x=x*y
/=	division, then assignment	x/=y	x=x/y
%=	remainder, then assignment	x%=y	x=x%y

Similar to increment and decrement, assignment operators are only shortcut notations (in the style of **C** and **C++**).

Conditional operator

- Syntax:

condition ? expression1 : expression2

- Semantics: comparable to

if (condition) sequence1 else sequence2

but at the level of expressions!

- The conditional operator is (only) meaningful in expressions.
- x = (y > 0 ? y : -y)** operates as

```
1  if (y > 0) x = y; else x = -y;
```

- x = y + (z > y ? 2 * y + z : y + z)** operates as

```
1  if (z > y)
2      w = 2 * y + z;
3  else
4      w = y + z;
5  x = y + w;
```

Bit operators

- Read and write access to single bits.
- Defined only for integer types and char.

Java operator	spoken as	precedence	order
unary, i.e., one operand			
<code>~</code>	bitwise 'not'	1	$R \rightarrow L$
binary, i.e., two operands			
<code>&</code>	bitwise 'AND'	3	$L \rightarrow R$
<code> </code>	bitwise 'OR'	5	$L \rightarrow R$
<code>^</code>	bitwise 'XOR'	4	$L \rightarrow R$

- Operations on individual bits analogously to logic truth tables:

a	$\sim a$	a	b	a&b	a b	a^b
1	0	1	1	1	1	0
0	1	1	0	0	1	1
		0	1	0	1	1
		0	0	0	0	0

- a^b** named as 'exclusive or'
- this results, for example, in operations on bytes:

A	00110011
B	01010101
$\sim A$	11001100
A&B	00010001
A B	01110111
A^B	01100110

Example: conversion of numbers into binary form using bit operators

```
1 public class K3B21E_Binary {
2     public static void main(String[] args) {
3
4         byte a, b = (byte) 1, c;
5         a = (byte) Integer.parseInt(args[0]);
6
7         System.out.println("decimal: " + a);
8
9         String binary= "";
10        while (true) {
11            c = (byte) (a & b);
12            if (c != 0) binary = "1" + binary;
13            else      binary = "0" + binary;
14            if (b == -128) break;
15            b = (byte) (b * 2);
16        }
17
18        System.out.println ("binary:  "+binary);
19    }
20 }
```

b serves as a mask to access the individual bits of **a** from right to left.

Example: execution of the loop for ***a* = 52**:

```
1  while (true) {  
2      c = (byte) (a & b);  
3      if (c != 0) binary = "1" + binary;  
4      else      binary = "0" + binary;  
5      if (b == -128) break;  
6      b = (byte) (b * 2);  
7  }
```

step	a	b	c	binary
1	00110100	00000001	00000000	0
2	00110100	00000010	00000000	00
3	00110100	00000100	00000100	100
4	00110100	00001000	00000000	0100
5	00110100	00010000	00010000	10100
6	00110100	00100000	00100000	110100
7	00110100	01000000	00000000	0110100
8	00110100	10000000	00000000	00110100

Shift operators

- 'shifting' access to a group of bits.
- defined only for integer types and char, together with a natural number

Java operator	spoken as	precedence
<<	left shift	2
>>	right shift with sign	2
>>>	right shift with zero fill	2

shift operations on bytes:

bits	00110110	bits	10011011
bits << 1	01101100	bits << 1	00110110
bits >> 1	00011011	bits >> 1	11001101
bits >> 5	00000001	bits >> 5	11111100
bits >>> 1	00011011	bits >>> 1	01001101
bits >>> 5	00000001	bits >>> 5	00000100

- With '>>' the sign is preserved,
- With '>>>' negative numbers turn positive...

Example for shift operators on `int`:

```
1 public class K3B22E_ShiftInt {
2     public static void main(String[] args) {
3
4         int a, b, c, d;
5         a = Integer.parseInt(args[0]);
6
7
8         b = a << 2;
9         c = a >> 2;
10        d = a >>> 2;
11
12        System.out.println
13            ("a:_____ " + a
14             + "\na_<<_2:_" + b
15             + "\na_>>_2:_" + c
16             + "\na_>>>_2:_ " + d);
17    }
18 }
```

for example with input ***a* = -17**:

binary				decimal
1111	1110	1111	-17
1111	1011	1100	-68
1111	1111	1011	-5
0011	1111	1011	1073741819

Example for shift operators on `char` (unsigned!):

```
1 public class K3B23E_ShiftChar {
2     public static void main(String[] args) {
3
4         char ch;  short a, b, c, d;
5         a = (short) Integer.parseInt(args[0]);
6
7         ch = (char) a;
8         b = (short) (ch << 2);
9         c = (short) (ch >> 2);
10        d = (short) (ch >>> 2);
11
12        System.out.println
13            ( "a:_____ " + a
14            + "\nch_<<_2_:_" + b
15            + "\nch_>>_2_:_" + c
16            + "\nch_>>>_2_:_" + d);
17    }
18 }
```

for example with input ***a* = -17**:

binary				decimal
1111	1111	1110	1111	-17
1111	1111	1011	1100	-68
0011	1111	1111	1011	16379
0011	1111	1111	1011	16379

Bit operators **&** and **|** can also be applied to the type **boolean**.

```
1 public class K3B24E_BitOp {
2     public static void main(String[] args) {
3
4         int i = 3, j = 2;
5         if ((++i < ++j) && (i++ > j++))
6             {}
7         else
8             System.out.println("&_:_i:_ " + i + "_j:_ " + j);
9
10        i = 3; j = 2;
11        if ((++i < ++j) & (i++ > j++))
12            {}
13        else
14            System.out.println("&_:_i:_ " + i + "_j:_ " + j);
15    }
16 }
```

- With **&&** the execution terminates as soon as the result is fixed!
- With **&** the expression is always evaluated completely!
- (see later, when we have discussed methods...)

Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- **Methods**

Goal:

- Reusability of frequently used sequences of instructions

Typical approaches:

- methods, procedures, functions, subprograms, macros, ...

Remarks:

- Similar concepts exist in practically all programming languages.
- We use *subprograms* and *procedure* also as more general terms for the different approaches.
- Every executable Java program contains a method **main** which is executed when the program starts.

Terminology with methods:

```
1 public class K3B25E_Rectangle {
2
3     static int wd, ht;
4     static int perimeter (int width, int height){ definition of method 1
5         return 2 * width + 2 * height;
6     };
7
8     static int area (int width, int height){ definition of method 2
9         return width * height;
10    };
11
12    public static void main(String[] args) {
13        int per, ar;
14
15        wd = Integer.parseInt(System.console().readLine("width : "));
16        ht = Integer.parseInt(System.console().readLine("height: "));
17
18        per = perimeter(wd, ht); method call 1
19        ar = area (wd, ht); method call 2
20
21        System.out.println ("perimeter: " + per + "\narea: " + ar );
22        System.exit (0);
23    }
24 }
```

```

1  static(1) int(2) perimeter(3) (int width, int height(4) ) {
2      int per;
3      per = 2 * width + 2 * height;
4      return per(5);
5  }
6  ...
7  myPerimeter = perimeter(6) (wd, ht(7));

```

⁽¹⁾ modifier	static
⁽²⁾ return type	int
⁽³⁾ method name	perimeter
⁽⁴⁾ formal parameters	(int width, int height)
method head	static int perimeter (int width, int height)
method body	{int per; ...; return per;}
⁽⁵⁾ return value	return per;
⁽⁶⁾ method call	perimeter (wd, ht)
⁽⁷⁾ actual parameters	wd, ht

- Usually a procedure executes a closed algorithm for *changing* input data.
- In programming languages input and output data are exchanged between caller and procedure in different ways:
 - ▶ through parameters
(input and output values)
 - ▶ through function values
(only output values, only functions)
 - ▶ through global variables
(input and output values)

formal parameters / actual parameters:

In the declaration of the method, the formal parameters are placeholders for the actual parameters given when calling the method:

```
1  static int perimeter (int width, int height){ formal parameters
2      return 2 * width + 2 * height; } return value
3      ...
4  perimeter (wd, ht); actual parameters
```

- Input parameters can be given as expressions (see below).
- The value of an input parameter defines the initial value of the corresponding formal parameter.
- Actual parameters must be compatible to the formal parameters.

Parameter passing:

Techniques to pass actual parameters to subprograms:

- **Call by reference** (for input and output parameters)
Passes the address of the variable such that the name of the formal parameter identifies the storage space of the actual parameter during the execution of the procedure.
- **Call by value** (for input parameters)
Evaluation of the actual parameter and assignment of this value to the corresponding formal parameter.
- **Call by result** (for output parameters)
Evaluation of the return value and assignment of this value to the corresponding actual parameter.
- **Call by name** (for input and output parameters)
The formal parameter is exchanged by the actual parameter in the program text. During the execution of the procedure every instance of the actual parameter must be newly evaluated.

Execution of **Call by value**:

- With **Call by value** the actual parameters are first copied to newly allocated memory locations for the formal parameters.
- Only the formal parameters are changed in the procedure (i.e., only the copies!).
- Modifications of the formal parameters thus do not change the actual parameters (but: see later!).
- If large objects are passed this way, this is very costly in terms of computational time (the actual copying) and memory usage (data is stored redundantly).

Consider the following procedure:

```
1 int doSomething (int hugo, int oskar) {  
2 //assumption: call using Call by value...  
3     hugo = hugo - 11;  
4     oskar ++;  
5     return hugo + oskar;  
6 }
```

```
1 value = doSomething ( a, b );
```

changes **value**, but not **a** or **b**.

Since we copy the data, actual parameters may be expressions, e.g.,

```
1 value = doSomething (3*a, b+a);
```

Execution of **Call by reference**:

- Instead of copying the data, references to the data are passed.
- Thus the formal parameter only states where in memory the actual parameter is stored.
- Every change of the formal parameter thus changes the actual parameter.

```
1 int doSomething (int hugo, int oskar) {  
2 //assumption: call using Call by reference ...  
3     hugo = hugo - 11;  
4     oskar ++;  
5     return hugo + oskar;  
6 }
```

```
1 value = doSomething ( a, b );
```

changes **value**, and also **a** and **b**!

The actual parameters must not be expressions!

Parameter passing in Java

- Input parameters of primitive type (**byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**) are passed using **Call by value**.
- Output parameters of primitive type (**byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**) are returned as a function value (mostly **Call by result**, the return value can be directly used in an expression)
- Objects (including arrays and String) are identified using reference variables (see later). A method is passed only the value of the corresponding reference variable (using **Call by value**). This '**Call by reference-value**' is suitable for input and output parameters.
- Remark: **Call by reference** would pass a reference to the reference variable. One could then change it such that it referenced a new object.

Function values:

- Functions return their result in form of a function value.
- Functions can be called directly in procedures.
- The type of the function determines the type of the return parameter.

Example declaration:

```
1  int perimeter (int width, int height) {  
2      return 2 * width + 2 * height; }  
3  
4  int area (int width, int height) {  
5      return width * height; }
```

Example calls:

```
1  int w = 7, h = 5;  
2  y = perimeter (w, h);  
3  x = 3 * area (5, y / 4);  
4  if (area (5, 12) == perimeter (12,18)) {...}
```

void functions:

Functions that do not return a value ('type' **void**) are called 'stand-alone'.

Example:

- declaration:

```
1  void perimeter (int width, int height)  {  
2      int per;  
3      per = 2 * width + 2 * height;  
4      System.out.println(per);  
5  }
```

- call:

```
1  perimeter (7,5);
```

Local variables and global variables:

A variable

- is local in the block B where it is declared
- is global in all inner blocks of B
- is unknown in all blocks outside of B
- is known from the point of its declaration to the end of the corresponding block
- ceases to exist when its local block is left

A procedure does not have its own internal memory, all locally defined variables cease to exist once the procedure ends.

If a procedure is executed repeatedly, all required information must be passed as input parameters!

- If a procedure changes a global variable, we call this a **side effect**. Example: In Java a method can access all variables known in the surrounding class (**class**).
- In some programming languages, global variables can be hidden by local variables, e.g.,

```
1  int x;  
2  int z (int y) {  
3      int x;  
4      x = 5*y;    //  outer x remains unchanged  
5      return x+y;  
6  }
```

Examples for side effects,
i.e., changing global variables in a procedure:

```
1 public class K3B26E_Sideeffect {
2
3     static int x,y;
4
5     static void swap () {
6         int help;
7         help = x;    // x and y are global !
8         x = y;
9         y = help;
10    }
11
12    public static void main(String[] args) {
13
14        x = Integer.parseInt(System.console().readLine("x:_"));
15        y = Integer.parseInt(System.console().readLine("y:_"));
16
17        System.out.println("before:  x= " + x + " _ _y=_ " + y );
18
19        swap ();
20
21        System.out.println("after: x= " + x + " _ _y=_ " + y );
22    }
23 }
```

Visibility in Java: Variables from outer blocks may not be 'overwritten' in Java!

```
1 public class K3B27E_Block {
2     public static void main(String[] args) {
3         int a = 3, b = 4;
4         { int c = 5, d = 6; }
5         { int c = 6, e = 7;
6             { int a = 2, f = 1;
7                 int c = 9;
8             }
9         }
10    }
11 }
```

error messages from the compiler:

```
1 K3B27_Block.java:9: error: variable a is already defined ...
2     { int a = 2, f = 1;
3         ^
4 K3B27_Block.java:10: error: variable c is already defined ...
5         int c = 9;
6         ^
7 2 errors
```

(This would be allowed, for example, in C++!)

Methods in Java:

- Java allows only functional procedures, called 'methods'.
- Parameter passing in and out of methods:
 - ▶ Input:
 - ★ parameters
Call by value (variables, primitive types)
'Call by reference-value' (referenced types)
 - ★ global variables (can be bad programming style...)
 - ▶ Output:
 - ★ function value
Call by result
 - ★ parameters
'Call by reference-value' (referenced types)
 - ★ global variables (can be bad programming style...)

- The **return** instruction causes the end of the procedure. Any method can include an arbitrary number of **return** instructions (including none).
- Local variables have no value when they are declared.
- Methods can be overloaded. (see later...)
- Methods can call themselves (recursive methods). (see later...)
- Methods are defined within classes. (see later...)
- Methods cannot be defined within other methods, 'local' methods exist only with restrictions.

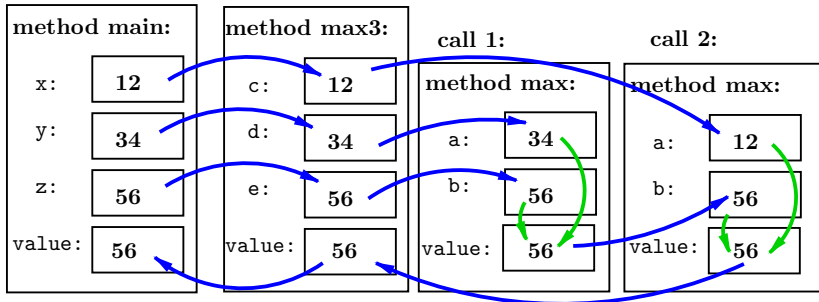
Method calls can be nested arbitrarily deep:

```
1 public class K3B28E_Nesting {
2
3     static int max (int a, int b)  {
4         return a > b ? a : b;
5     }
6     static int max3 (int c, int d, int e)  {
7         return max (c, max (d, e) );
8     }
9
10    public static void main(String[] args) {
11
12        int x, y, z;
13        x = Integer.parseInt (args[0]);
14        y = Integer.parseInt (args[1]);
15        z = Integer.parseInt (args[2]);
16
17        System.out.println("Maximum:_" + max3 (x, y, z) );
18
19    }
20 }
```

```

1  ...
2  static int max (int a, int b) {
3      return a > b ? a : b;
4  }
5  static int max3 (int c, int d, int e) {
6      return max (c, max (d, e) );
7  }
8  public static void main(String[] args) {
9      int x, y, z;
10     ... max3 (x, y, z) ) ...

```



Recursive methods

- A method/procedure is called **recursive** when it calls itself.
- At runtime a separate memory area is reserved for every call (for parameters, local variables, etc).
- With recursive methods it is often very simple to implement problems defined in a recursive way.
- Example 1, factorial of n :

$$0! = 1$$

$$n! = n \cdot (n - 1)!$$

- Example 2, Fibonacci numbers:

$$\textit{fib}(0) = 0$$

$$\textit{fib}(1) = 1$$

$$\textit{fib}(n) = \textit{fib}(n-1) + \textit{fib}(n-2) \text{ for } n > 1$$

```
1 public class K3B29E_Factorial {
2
3     static long factorial (int n) {
4         if (n == 0) return 1;
5         else return (n * factorial (n-1));
6     }
7
8     public static void main (String[] args) {
9         for (int k=0; k<22; k++)
10             System.out.println(
11                 "Factorial of " + k + " is: " + factorial (k) );
12     }
13 }
```

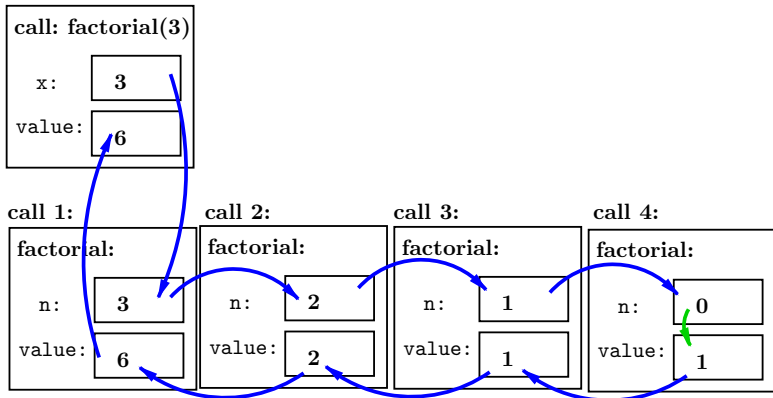
(The values up to **20! = 2.432.902.008.176.640.000** are correct, **21!** is too large for **long**.)

```

1  static long factorial (int n) {
2      if (n == 0) return 1;
3      else return (n * factorial (n-1));
4  }

```

Evaluation: “recursive descend”



```
1 public class K3B30E_Fibonacci {
2
3     static long fib (int n)  {
4         if (n==0) return 0;
5         else
6             if (n==1) return 1;
7             else return ( fib (n-1) + fib (n-2) );
8     }
9
10    public static void main(String[] args) {
11        int k= Integer.parseInt(args[0]);
12        System.out.println(
13            "Fibonacci_" + k + "_" + fib (k) );
14    }
15 }
```

Effort of the recursive solution:

- ***fib*(40) = 102 334 155**
(requires 267 914 295 calls, about 0.8s on PC)
- ***fib*(45) = 1 134 903 170**
(requires 2 971 215 072 calls, about 8.4s)
- ***fib*(90) = 2 880 067 194 370 816 120**
(requires 7 540 113 804 746 346 428 calls...)
with 350 000 000 calls/s this takes about **$2,1 \cdot 10^9$** s (about 66 years!)
- Conclusion: not every solution is a good (= efficient) solution...
- Problem:
 - ▶ non-linear recursion
 - ▶ repeated repetition of calls
(especially ***fib*(0)**, ***fib*(1)**)

Fibonacci numbers can also be expressed via the 'golden ratio':

$$fib(n) = \frac{\Phi^n - \Psi^n}{\sqrt{5}}$$

with $\Phi = \frac{1+\sqrt{5}}{2} = 1.61803\dots$ and $\Psi = \frac{1-\sqrt{5}}{2} = -0.61803\dots$

Since $|\Psi| < 1$ we have $\frac{|\Psi|^n}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}$

thus

$$|fib(n) - \frac{\Phi^n}{\sqrt{5}}| \leq \frac{1}{2}$$

and therefore:

$$fib(n) = \lfloor \frac{\Phi^n}{\sqrt{5}} + \frac{1}{2} \rfloor$$

(where $\lfloor x \rfloor$: greatest integer less or equal to x)

Fibonacci numbers are thus growing exponentially:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,

Let **$ra(n)$** be the number of (recursive) calls of **$fib(n)$** , then:

$$\mathbf{ra(n) = ra(n-1) + ra(n-2) + 1}$$

The number of calls grows similarly to the Fibonacci numbers themselves (even a little faster), i.e., exponentially.

Computation of Fibonacci numbers with “dynamic programming”

- partial results are stored (\leadsto array)

```
1 public class K3B31E_Fibonacci_dynamic {
2
3     static long fib (int n)  {
4
5         long[] f = new long[n+1];
6
7         if (n <= 1) return n;
8         f[0] = 0;
9         f[1] = 1;
10        for (int i = 2; i <= n; i++ ) {
11            f[i] = f[i-1] + f[i-2];
12        }
13        return f[n];
14    }
15
16    public static void main(String[] args) {
17        int k= Integer.parseInt(args[0]);
18        System.out.println(
19            "Fibonacci_" + k + "_" + fib (k) );
20    }
21 }
```

Reduction to partial results that are still needed:

- Only two old values are still used...

```
1 public class K3B32E_Fibonacci_iterative {
2
3     static long fib (int n)  {
4         if (n <= 1) return n;
5         long fib_0 = 1, fib_1 = 0, fib_2;
6         for (int i = 2; i <= n; i++ ) {
7             fib_2 = fib_1;
8             fib_1 = fib_0;
9             fib_0 = fib_1 + fib_2;
10        }
11        return fib_0;
12    }
13
14    public static void main(String[] args) {
15        int k= Integer.parseInt(args[0]);
16        System.out.println(
17            "Fibonacci_" + k + " = " + fib (k) );
18    }
19 }
```

Recursion vs. iteration

- The example of the Fibonacci numbers shows that an 'elegant' algorithm is not necessarily efficient.
- The efficiency of recursive algorithms often depends on how well the compiler can optimize them.
- It is always possible to reformulate recursive algorithms as iterative algorithms,
in 'more complex' cases this requires the data structure 'stack' (see later).

Predefined methods

- The Java Software Development Kit (SDK) includes a large number of predefined methods.
- J2SE 5.0 includes about 3000 methods in more than 3000 classes.
- Example: class **Math** (`java.lang.Math`), see <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

constants **$e = 2.71828182846...$** and **$\pi = 3.14159265359...$**

```
1    public static final double E
2    public static final double PI
```

methods:

```
1    public static double ceil (double a)
2    public static double floor (double a)
3    public static double rint (double a)
4
5    public static long round (double a)
6    public static int round (float a)
```

- Predefined methods can be directly used without declaration.
- **Math** belongs to the package `java.lang` (see later) and thus does not have to be imported.
- All methods in **Math** are declared as **static** (see later), they are called using `classname.methodname`, for example `Math.ceil (3.14);`

```
1 public class MathCall {  
2  
3     public static void main (String[] args) {  
4  
5         System.out.println( "square root of (e * pi) : "  
6             + Math.sqrt ( Math.E * Math.PI ) );  
7     }  
8 }
```

More methods from `java.lang.Math`:

```
1 public static double sin (double angle) // sine
2 public static double cos (double angle) // cosine
3 public static double tan (double angle) // tangent
4
5 public static double exp (double x) //  $e^x$ 
6 public static double log (double x) // logarithm  $\ln x$ 
7 public static double pow (double x, double y) //  $x^y$ 
8
9 public static double random () // random number between 0.0 and 1.0
10
11 public static double sqrt (double x) // square root of x
```

`log`, `pow` and `sqrt` yield **NaN** in the case of an error.


```
1 public static double abs (double x)
2 public static float abs (float x)
3 public static long abs (long x)
4 public static int abs (int x)
5
6 public static double max (double x, double y)
7 public static float max (float x, float y)
8 public static long max (long x, long y)
9 public static int max (int x, int y)
10
11 public static double min (double x, double y)
12 public static float min (float x, float y)
13 public static long min (long x, long y)
14 public static int min (int x, int y)
15 ...
16 ...
```

abs, **max**, **min** and **round** are examples for *overloaded methods*.

Method Overloading

```
1 public class K3B33E_Overloading {
2     public static void main (String[] args) {
3
4         double db = -6.12345678908642;
5         float fl  = -6.1234567F;
6         long lg   = -12345678908642L;
7         int in    = -12345678;
8
9         System.out.println(
10             "abs_(db)_" + Math.abs ( db )
11             + "\nabs_(fl)_" + Math.abs ( fl )
12             + "\nabs_(lg)_" + Math.abs ( lg )
13             + "\nabs_(in)_" + Math.abs ( in ) );
14     }
15 }
```

- An operator is *overloaded* when it has multiple meanings for the same syntax, i.e., when it can be used for different tasks.
- “-” as an example for an overloaded operator in elementary mathematics:

	unary sign operator	binary sub- traction operator
natural numbers	-	-
integers	-	-
rational numbers	-	-
real numbers	-	-

- Some languages (e.g., **C++**) allow to define operators for new data types (“operator overloading”)
- In **Java** “operator overloading” is not possible.

- A method is overloaded when variants of this method with the same name exist, that perform a different task and thus have a different semantics. (Examples: **abs**, **max**, **min** in **java.lang.Math**).
- Variants of overloaded methods have the same name, but must have different parameter lists, i.e., a different number of parameters and/or different parameter types.
- Thus the *signature* of a method is the combination of return type, method name and parameter list.
- The declaration of two methods whose signatures differ only in the return type yields a compiler error message.
- Overloaded methods are typical for object-oriented programming languages, since here the operators for the different classes of objects are implemented as methods (see later).

Local **static** methods can have the same signature as global **static** methods, without overwriting them.

```
1 public class K3B34E_local_Max {
2
3     static int max (int x, int y)  {
4         return x > y ? x : y;
5     }
6
7     public static void main(String[] args) {
8         System.out.println (
9             "Maximum:_" + max (3, 7)          + "_ (local method!)"
10            + "\nMaximum:_" + Math.max (3, 7) + "_ (Math.max)"
11        );
12    }
13 }
```

The global **static** method can always be uniquely identified using the class name, e.g. **max** (3,7) vs. **Math.max** (3,7).

4 Non-primitive data types in Java

- Strings
- Arrays
- StringTokenizer
- StringBuffer

String from `java.lang.String`

- is *not* a basic type like `int`, `float` etc.
- is a class (**class**)
- belongs to the package `java.lang` (i.e., no import required)
- can sometimes be used like a basic type

Declaration / copy:

```
1 String str1;                //declaration of a variable
2
3 String str2 = "a String";   //declaration with initialization
4
5 str1 = str2;                //copy of the reference variable
6
7 char text[] = {'a', 'b', 'c'};
8 str1 = new String(text);    //redeclaration with initialization
```

'**str1**', '**str2**' are reference variables that can only refer to objects from the class **String** (see later...).

- Methods in String are NOT **static**.
- We thus need to refer the object on which the method should be applied.

Example: **public int length ()**

```
1 public class K4B01E_StringLength {
2
3     public static void main (String[] args) {
4
5         String st1 = "a String";
6         String st2 = "another String";
7
8         System.out.println("\"" + st1 + "\":_length " + st1.length() );
9         System.out.println("\"" + st2 + "\":_length " + st2.length() );
10        st2 = st1;
11        System.out.println("\"" + st2 + "\":_length " + st2.length() );
12    }
13 }
```


Extraction of single characters and substrings

Consider `String aString = "This is a String";`

T	h	i	s		i	s		a		S	t	r	i	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- `public char charAt (int index)`

yields character at position **index** (from 0 to **length-1**)

`aString.charAt (3);`

~> 's'

- `public String substring (int beginIndex)`

yields substring from **beginIndex** to end of String

`aString.substring (10);`

~> "String"

- `public String substring (int beginIndex, int endIndex)`

yields substring from **beginIndex** to **endIndex-1**

`aString.substring (0, 4);`

~> "This"

T	h	i	s		i	s		a		S	t	r	i	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- `public int indexOf (char ch)`

yields first position of **ch** (forward search starting at 0)

`index = aString.indexOf ('i');` \leadsto `index = 2`

- `public int indexOf (char ch , int fromIndex)`

yields next position of **ch** (forward search starting at **fromIndex**)

`index = aString.indexOf ('i' , index + 1);` \leadsto `index = 5`

- `public int lastIndexOf (char ch)`

yields last position of **ch** (backward search starting at **length-1**)

`index = aString.lastIndexOf ('i');` \leadsto `index = 13`

- `public int lastIndexOf (char ch , int fromIndex)`

yields next position of **ch** (backward search starting at **fromIndex**)

`index = aString.lastIndexOf ('i' , index-1);` \leadsto `index = 5`

Localizing substrings in a String

T	h	i	s		i	s		a		S	t	r	i	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- `public int indexOf (String str)`

yields first position of **str** (forward search starting at 0)

`index = aString.indexOf ("is");` \leadsto `index = 2`

- `public int indexOf (String str, int fromIndex)`

yields next position of **str** (forward search starting at **fromIndex**)

`index = aString.indexOf ("is" , index + 1);` \leadsto `index = 5`

- `public int lastIndexOf (String str)`

yields last position of **str** (backward search starting at **length-1**)

`index = aString.lastIndexOf ("his");` \leadsto `index = 1`

- `public int lastIndexOf (String str, int fromIndex)`

yields next position of **str** (backward search starting at **fromIndex**)

`index = aString.lastIndexOf ("his" , index-1);` \leadsto `index = -1`

Manipulation of Strings

Functions with String as return type:

- The input String is **not** modified,
- the manipulations result in new Strings
- Strings are **immutable**, i.e., they cannot be modified

```
1 public String replace (char oldChar, char newChar)
2 public String toLowerCase ( )
3 public String toUpperCase ( )
4 public String trim ( )
5 public String concat (String str)
```

Example with **String aString = "a String"**

```
1 aString.replace ('i', 'u')  ~> "a Strung"
2
3 aString.toLowerCase ()  ~> "a string"
4
5 aString.toUpperCase ()  ~> "A STRING"
6
7 "___a String    ".trim ()  ~> "a String"
8
9 "extend ".concat (aString)  ~> "extend a String"
10 aString.concat (" is extended")  ~> "a String is extended"
```

String comparisons

```
1 public boolean startsWith (String prefix)
2 public boolean startsWith (String prefix, int index)
3 public boolean endsWith (String suffix)
4 public boolean regionMatches (int cindex,
5                               String str, int strindex, int size)
6 public boolean regionMatches (boolean case, int cindex,
7                               String str, int strindex, int size)
```

Example with `String aString = "This is a String"`

```
1 aString.startsWith ("this")           ~> false
2 aString.startsWith ("is", 5)           ~> true
3 aString.endsWith ("ing")               ~> true
4
5 aString.regionMatches (13, "KRINGEL", 2, 3) ~> false
6 aString.regionMatches (true, 13, "KRINGEL", 2, 3) ~> true
7                                     // ignore case
```

String equality

```
1 public boolean equals (String anotherString)
2 public boolean equalsIgnoreCase (String anotherString)
```

Example with **String aString = "This is a String"**

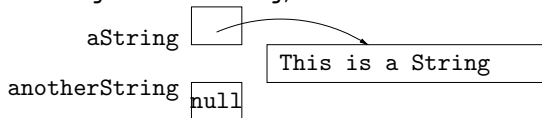
```
1 aString.equals ("this is a string")           ~> false
2 aString.equalsIgnoreCase ("this is a string") ~> true
3 "This is a String".equals(aString)           ~> true
```

many more methods for **String**: see the Java documentation

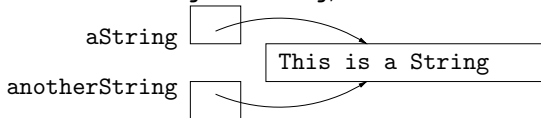
Reference variables for Strings

- String aString = "This is a String";

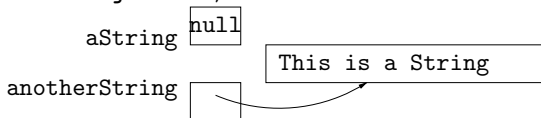
String anotherString;



- anotherString = aString;



- aString = null;



- Reference variables do not store the actual data, but only a reference to the data.
- A comparison of two reference values (with “==”) only compares the references, not the referenced data.
- An assignment between reference variables only copies the reference, not the actual data.
(Since Strings are immutable, copying a String would not make much sense anyway...)

Preview of the concept of classes in Java

```
1    str = new String ( );  
2    str = new String ( "a String");  
3    char array[] = {'a', 'b', 'c'};  
4    str = new String ( array );
```

- **String()**, **String(String str)**, **String(char[] chArray)** are *constructors* of the class **java.lang.String**.
- **str = new String (array)** causes:
 - ▶ generation of a new *object* and initialization with the values from the char array, i.e., as **"abc"**.
 - ▶ assignment of a reference to the object to the variable **str**.

Attention:

- Do **not** use “==” to compare Strings!
- This compares only the references, not the Strings!

```
1 public class K4B02E_StringEqual {
2     public static void main(String[] args) {
3         String g = "Hello", h = "Hello";
4         String i = "Hello" + "";
5         String j = h + "";
6
7         System.out.println( "1." + g.equals(h) );
8         System.out.println( "2." + g.equals(i) );
9         System.out.println( "3." + g.equals(j) );
10        System.out.println( "4." + ( g == h ) );
11        System.out.println( "5." + ( g == i ) );
12        System.out.println( "6." + ( g == j ) );
13        System.out.println( "7." +      g == h      );
14    }
15 }
```

- 1./2./3.: content, i.e, true, since same content
- 4./5.: references, true, created at compilation time (“String pool”)
- 6.: references, false, since not in the “String pool”
- 7.: references, but ‘+’ stronger than ‘==’

Test for palindrome as an example for String manipulations:

```
1 public class K4B03E_Palindrom {
2     /* tests if a given string is a palindrome */
3
4     public static void main(String[] args) {
5
6         String str = System.console().readLine
7             ("input test string: ");
8
9         int left, right;
10        for ( left = 0, right = str.length () - 1;
11            left < right;
12            left++, right --) {
13            if ( str.charAt (left) != str.charAt (right) ) break;
14        }
15
16        System.out.println("palindrome " + (left >= right) );
17    }
18 }
```

Reversing as an example for String manipulations:

```
1 public class K4B04E_Reverse {
2     /*  reverses the input string      */
3
4     public static void main(String[] args) {
5
6         String str = System.console().readLine("input string: ");
7
8         String reverse="";
9
10        for (int index =0; index < str.length(); index++ ) {
11            reverse = str.charAt(index) + reverse;
12        }
13
14        System.out.println( str + "_reversed " + reverse );
15    }
16 }
```

With the (overloaded) static method **valueOf** (...) it is possible to convert objects of many classes to Strings:

```
1 public class K4B05E_StringValueOf {
2     public static void main( String args[] ) {
3         char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
4         boolean booleanValue = true;
5         char characterValue = 'Z';
6         int integerValue = 7;
7         long longValue = 10000000L;
8         float floatValue = 2.5f;
9         double doubleValue = 33.333;
10
11         System.out.println(
12             "char_array:_ " + String.valueOf( charArray ) +
13             "\nboolean:_" + String.valueOf( booleanValue ) +
14             "\nchar:_" + String.valueOf( characterValue ) +
15             "\nint:_" + String.valueOf( integerValue ) +
16             "\nlong:_" + String.valueOf( longValue ) +
17             "\nfloat:_" + String.valueOf( floatValue ) +
18             "\ndouble:_" + String.valueOf( doubleValue ) );
19     }
20 }
```

(this uses **toString()** of the corresponding classes)

4 Non-primitive data types in Java

- Strings
- **Arrays**
- StringTokenizer
- StringBuffer

- Arrays have existed already in the first high-level programming languages to represent vectors, matrices, etc.
- Usually an array is an aggregation of multiple data elements of the same type to a data structure.
- In Java an array is a group of variables or reference variables of the same type.
- In Java this aggregation can be repeated over multiple steps.
- In other programming languages, multi-dimensional arrays form a unit (in the memory of the computer)
Java always uses multiple levels (arrays of arrays of ...)
- Once an array was initialized in Java, its size cannot be changed, the number of its elements is fixed.
(but: the elements can be reference variables...).

```
1 public class K4B06E_Days {
2     public static void main (String[] args) {
3
4         //declaration and initialization of an array of dimension 1
5         //name: week,
6         //type of elements: String
7         //number of elements: 7
8
9         String [] week = { "Monday", "Tuesday", "Wednesday",
10                            "Thursday", "Friday",
11                            "Saturday", "Sunday"};
12
13         // the indexes of the array elements run from 0 to 6
14         // (from 0 to week.length-1)
15
16         System.out.println( "Days of the week:" );
17         for (int i = 0; i < week.length; i++)
18             System.out.println("_week [" + i + "]_=_ " + week[i]);
19         // week [i]: access the ith array element
20
21     }
22 }
```


- The indexes for the elements of an array **anArray** run from 0 to **anArray.length-1**.
- If an index value < 0 or $\geq \text{length}$ is accessed, an **ArrayIndexOutOfBoundsException** is generated.
- An index value must be a non-negative integer or an integer expression, e.g,
anArray [j++ + a[0]] or
anArray [(int)Math.pow(2,2)]

Arrays are used in three steps:

- *declaration of the variable* (including assignment of its type)

Ex: `int [] iArray;`

`iArray` is a variable referencing an array with elements of type `int`.

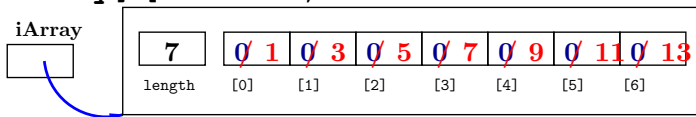
- *allocation of memory & initialization*

Ex: `iArray = new int [7];`

allocation of memory for 7 elements of type `int`
(automatically initialized with value 0)
and a memory cell for the array size.

- *value assignment*

Ex: `for (int i=0; i<iArray.length; i++)`
`iArray[i] = 2*i+1;` with values:



- When memory is allocated, the array is already filled with default values.
- Examples for default values:
 - ▶ `int [] intArray = new int [4];`
~> 0 0 0 0
 - ▶ `float [] floatArray = new float [4];`
~> 0.0 0.0 0.0 0.0
 - ▶ `boolean [] booleanArray = new boolean [4];`
~> false false false false
 - ▶ `char [] charArray = new char [4];`
~> 0x0000 0x0000 0x0000 0x0000
 - ▶ `String [] StringArray = new String [4];`
~> null null null null

Example: arrays as parameters

Arrays are passed using *call by reference-value*:

```
1 public class K4B07E_TestFeld {
2
3     static void showArray (int [] anArray) {
4         for (int i = 0; i < anArray.length; i++)
5             System.out.println(i + ":_:" + anArray [i]);
6         System.out.println();
7     }
8
9     public static void main (String[] args) {
10         int [] testArray;
11
12         testArray = new int [5];
13         for (int i = 0; i < 5; i++) testArray [i] = i + 1;
14         showArray (testArray);
15
16         testArray = new int [] { 11, 22, 33, 44 };
17         showArray (testArray);
18     }
19 }
```

Example: evaluation of an int expression via the command line:

```
1 public class K4B08E_CLArgs {
2     public static void main( String args[] ) {
3         int result;
4         int arg1 = Integer.parseInt ( args [ 0 ] );
5         char arg2 = args[1].charAt(0);
6         int arg3 = Integer.parseInt ( args [ 2 ] );
7         switch (arg2) {
8             case '+': result = arg1 + arg3; break;
9             case '-': result = arg1 - arg3; break;
10            case '*': result = arg1 * arg3; break;
11            default : result = 0;
12        }
13        System.out.println(
14            arg1 + "_" + arg2 + "_" + arg3 + "_=" + result);
15    }
16 }
```

java K4B08E_CLArgs 77 + 23 ~> 100

java K4B08E_CLArgs 129 - 32 ~> 97

java K4B08E_CLArgs 32 "*" 32 ~> 1024

Example: sum of the command line parameters

```
1 public class K4B09E_CLSum {  
2     public static void main( String args[] ) {  
3  
4         int sum = 0, i;  
5  
6         for (i = 0; i < args.length; i++)  
7             sum = sum + Integer.parseInt ( args [i] );  
8  
9         System.out.println(  
10             "The sum of the " + i + "_elements is: " + sum);  
11     }  
12 }
```

java K4B09E_CLSum 234 345 456 567 678 ~> 2280

Example: distribution of the number of pips of a die

```
1 import javax.swing.*;
2 public class K4B10E_Dice {
3     public static void main( String args[] ) {
4         int frequency[] = new int[ 8 ];
5
6         for ( int roll = 1; roll <= 300; roll++ )
7             frequency [ 1 + (int) ( Math.random() * 6 ) ] ++;
8
9
10        String output = "number of pips\tfrequency\tistogram";
11        for ( int side = 0; side < frequency.length; side++ ) {
12            output += "\n" + side + "\t" + frequency[side] + "\t";
13            for ( int k = 0; k < frequency[ side ]; k++ )
14                output += "*";
15        }
16
17        JTextArea outputArea = new JTextArea();
18        outputArea.setText( output );
19        JOptionPane.showMessageDialog( null, outputArea,
20            "300_rolls yield:", JOptionPane.INFORMATION_MESSAGE );
21    }
22 }
```

300 throws yield:



number of pips	frequency	histogram
0	0	
1	63	*****
2	51	*****
3	66	*****
4	35	*****
5	43	*****
6	42	*****
7	0	

OK

Example: sorting an array of type char []

```
1 import java.util.Scanner;
2 public class K4B11E_BubbleSort {
3     public static void main(String[] args) {
4
5         Scanner sc = new Scanner(System.in);
6         System.out.print("Test string: ");
7         String ts = sc.nextLine();
8         char [] chArray = ts.toCharArray();
9
10        System.out.println ( 0 + ":_ " + new String(chArray) );
11
12        for (int i = 1; i < chArray.length; i++){
13            for (int j = 0; j < chArray.length - i; j++){
14                if ( chFeld [j] > chArray [j+1]) {
15                    char help = chArray [j];
16                    chArray [j] = chArray [j+1];
17                    chArray [j+1] = help;
18                }
19                System.out.println ( i + ":_ " + new String(chArray) );
20            }
21        }
22    }
```

Exemplary executions:

```
1    for (int i = 1; i < chArray.length; i++){
2        for (int j = 0; j < chArray.length - i; j++)
3            if ( chArray [j] > chArray [j+1]) {
4                char help = chArray [j];
5                chArray [j] = chArray [j+1];
6                chArray [j+1] = help;
7            }
8        System.out.println ( i + ":_ " + new String(chArray) );
9    }
```

Test string: "one_String"

0: one_String
1: ne_Soringt
2: e_Snoingrt
3: _Seningort
4: S_eingnort
5: S_eignnort
6: S_eginnort
7: S_eginnort
8: S_eginnort
9: S_eginnort

Test string: "9876543210"

0: 9876543210
1: 8765432109
2: 7654321089
3: 6543210789
4: 5432106789
5: 4321056789
6: 3210456789
7: 2103456789
8: 1023456789
9: 0123456789

Sorting of an array in a method, parameter passing with '*call by reference-value*'

```
1 public class K4B12E_BubbleSort2 {
2
3     public static void sort ( int [] a) {
4         for (int i = 1; i < a.length; i++)
5             for (int j = 0; j < a.length - i; j++)
6                 if ( a[j] > a[j+1] ) {
7                     int t  = a[j];
8                     a[j]   = a[j+1];
9                     a[j+1] = t;
10                }
11    }
12
13    public static void main ( String [] args ) {
14        int [] array = new int [args.length];
15        for (int i = 0; i < args.length; i++)
16            array[i] = Integer.parseInt ( args [i] );
17
18        sort (array);
19
20        for (int k = 0; k < array.length; k++)
21            System.out.print(array [k] + "_");
22        System.out.println();
23    }
24 }
```

Multidimensional arrays:

```
1 public class K4B13E_Matrix {
2     public static void main(String[] args) {
3
4         int n = Integer.parseInt ( args [0] );
5         int m = Integer.parseInt ( args [1] );
6
7         int [][] matrix = new int [n] [m];
8
9         for ( int i = 0; i < matrix.length; i++ )
10             for ( int j = 0; j < matrix[i].length; j++ )
11                 matrix[i][j] = 10000 + i*100 + j;
12
13         for ( int i = 0; i < matrix.length; i++ ) {
14             for ( int j = 0; j < matrix[i].length; j++ )
15                 System.out.print(matrix[i][j] + "_");
16             System.out.println();
17         }
18     }
19 }
```

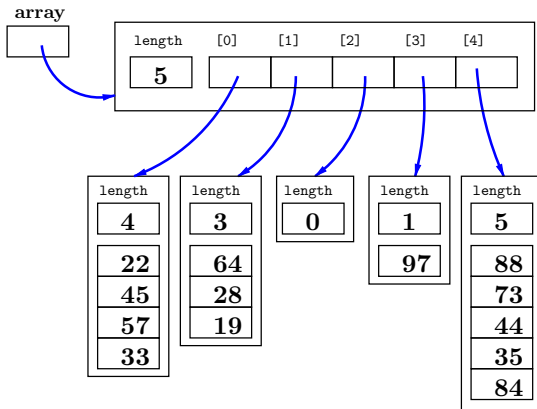
```
java K4B13E_Matrix 3 5 ~> 10000 10001 10002 10003 10004
                             10100 10101 10102 10103 10104
                             10200 10201 10202 10203 10204
```

The rows of multidimensional arrays can have different lengths:

```
1 public class K4B14E_ArrayArray {
2     public static void main(String[] args) {
3         int [][] array = {
4             {22, 45, 57, 33},
5             {64, 28, 19},
6             {},
7             {97},
8             {88, 73, 44, 35, 84}
9         };
10
11     System.out.println("number of rows "+ array.length);
12     for ( int i = 0; i < array.length; i++ ) {
13         System.out.print(i + "_[" + array[i].length + "]:_");
14         for ( int j = 0; j < array[i].length; j++ )
15             System.out.print(array [i][j] + "_");
16         System.out.println();
17     } } }
```

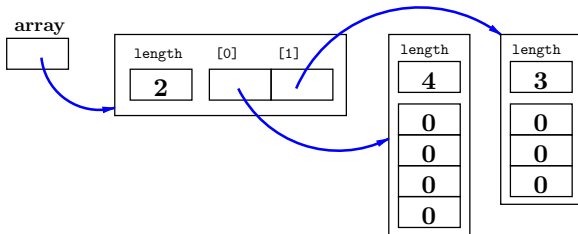
```
number of rows 5
0 [4] : 22 45 57 33
~> 1 [3] : 64 28 19
2 [0] :
3 [1] : 97
4 [5] : 88 73 44 35 84
```

```
1  int [] [] array = {  
2      {22, 45, 57, 33},  
3      {64, 28, 19},  
4      {},  
5      {97},  
6      {88, 73, 44, 35, 84}}
```



Stepwise memory allocation for multidimensional arrays:

```
1  int array[][];  
2  array = new int[ 2 ][ ];  
3  // memory allocation for the first dimension  
4  
5  array[ 0 ] = new int[ 4 ];  
6  // memory allocation for subarray 0  
7  
8  array[ 1 ] = new int[ 3 ];  
9  // memory allocation for subarray 1
```



Options for specifying the number of dimensions:

```
1 int [] a; // ok, one dimension
2 int b []; // ok, one dimension
3 int [] c, d, e; // ok, one dimension
4 int f [][]; // ok, two dimensions
5 int g [][][]; // ok, three dimensions
6 int [] oneD, twoD []; // ok
```

oneD has one dimension, but **twoD** has two dimensions!

Permitted order for partial allocation:

```
1 int [] [] [] a; // ok
2 a = new int [7] [] []; // ok
3 a = new int [7] [4] []; // ok
4 a = new int [7] [4] [6]; // ok
```

Illegal order for partial allocation:

```
1 a = new int [] [4] []; // wrong!
2 a = new int [] [4] [6]; // wrong!
3 a = new int [] [] [6]; // wrong!
4 a = new int [7] [] [6]; // wrong!
```


Excerpt from `java.util.Arrays`:

```
public static void sort ( int[] anArray )
```

- Example call: **`Arrays.sort (anArray);`**
- sorts **`anArray`** using Quicksort algorithm
- useful for huge arrays
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, **`double`**

```
public static boolean equals (int[] anArray, Object anObject)
```

- Example call: **`Arrays.equals (anArray, anotherArray);`**
- yields true if **`anotherArray`** is array of the same type as **`anArray`**, has the same length and the elements in identical positions have the same values
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, **`double`**

```
public static int binarySearch ( int[] anArray, int value )
```

- Example call:
`index = Arrays.binarySearch (anArray, 4711);`
- yields index of the element in **anArray** which contains the value **value**
- since 'binary search' is performed, **anArray** must be sorted
- if **value** is not found, a negative value **r** is returned;
— **r - 1** denotes the position where **value** would need to be inserted to keep the array sorted.
- interesting for very huge arrays
- analogously for **long**, **short**, **char**, **byte**, **boolean**, **float**, and **double**

from **java.lang.System**:

```
public static void arraycopy (Object src, int srcPos,  
                             Object dest, int destPos, int length)
```

- copies an excerpt of length **length**
from array **src** (starting at position **srcPos**)
to the array **dest** (starting at position **destPos**)
- Attention: If the array contains references, this copies only the
references, NOT the data!

from **java.lang.Object**:

```
protected Object clone() throws CloneNotSupportedException
```

- Example call (with required type cast):
int[][] anotherArray = (int[][])
anArray.clone();
- generates 'clone' (full copy) of an array
- Attention: If the array contains references, this copies only the
references, NOT the data!

Example program K4B15E_AsciiArt

2D graphics made of single characters, with animation,
width/height through command-line parameters (here 60x6)

```
X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
.X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
..X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
...X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
```

Uses the following initialization method from `java.util.Arrays`:

```
public static void fill ( int[] anArray, int value)
```

- Example call `Arrays.fill (anArray, 42);`
- sets all elements of `anArray` to the value `value`
- analogously for `long`, `short`, `char`, `byte`, `boolean`, `float`, `double`

```

1 import java.util.Arrays;
2 public class K4B15E_AsciiArt {
3     public static char [][] image;
4
5     public static void repaint(){
6         for (int row=0; row< image.length; row++)
7             System.out.println(new String(image[row]));
8         if ( ! "".equals(System.console().readLine("\n\n")))
9             System.exit(0);
10    }
11
12    public static void main(String[] args) {
13        int width = Integer.parseInt(args[0]);
14        int height = Integer.parseInt(args[1]);
15        image = new char [height][width];
16
17        for (int row=0; row< height; row++)
18            Arrays.fill (image[row],'.');
19
20        int x=0, y=0;
21        while (true) {
22            image[y][x]='X';
23            repaint();
24            x = (x + 1) % width;
25            y = (y + 1) % height;
26    } } }

```

Example program K4B16E_Snake

animated snake in 2D graphics,

width/height/length through command-line parameters (here 60/6/25)

```

      =           =
    = =         = =       W
  =   =       =   =     =
=   =   =   =   =   =
= =       = =       = =
    =           =       =
```

Here `repaint()` as before, but modified `main` method:

```
1  public static void main(String[] args) {
2      if (args.length != 3) {
3          System.out.println("call: java K4B16E_Snake "
4                               + "width height length");
5          return;
6      }
7
8      int width = Integer.parseInt(args[0]);
9      int height = Integer.parseInt(args[1]);
10     int length = Integer.parseInt(args[2]);
11     image = new char [height][width];
```

```
12  for (int row=0; row< height; row++)
13      Arrays.fill (image[row],'_');
14
15  int xa=0, ya=0, dxa=1, dya=1;
16  int xe=0, ye=0, dx=1, dy=1;
17  while (true) {
18
19      image[ya][xa]='W';
20
21      repaint();
22
23      image[ya][xa]='_';
24      xa = xa + dxa; if (xa>=width-1 || xa<=0) dxa = -dxa;
25      ya = ya + dya; if (ya>=height-1 || ya<=0) dya = -dya;
26      if (length > 0) {
27          length --;
28      } else {
29          image[ye][xe]='_';
30          xe = xe + dx; if (xe>=width-1 || xe<=0) dx = -dx;
31          ye = ye + dy; if (ye>=height-1 || ye<=0) dy = -dy;
32      }
33  }
34 }
```

Exercices on dealing with references on arrays:

(1) How does one compare one-dimensional arrays?

```
1 import java.util.Arrays;
2 public class K4B17E_ArrayEq_1_dim {
3     public static void main(String[] args) {
4
5         int [] a = new int [4];
6         int [] b = new int [4];
7         boolean eq;
8
9         // wrong:
10         eq = (a == b);
11         System.out.println("wrong: identical = " + eq );
12
13         // correct:
14         eq = Arrays.equals (a, b);
15         System.out.println("correct: identical = " + eq );
16     }
17 }
```


(2) How does one compare two-dimensional arrays?

```
1 import java.util.Arrays;
2 public class K4B18E_ArrayEq_2_dim {
3     public static void main(String[] args) {
4         int [][] a = new int [4][5];
5         int [][] b = new int [4][5];
6         boolean eq;
7
8         // wrong:
9         eq = Arrays.equals (a, b);
10        System.out.println("wrong: identical = " + eq );
11
12        // correct:
13        eq = true;
14        if (a.length != b.length ) {
15            eq = false;
16        } else {
17            for (int i = 0; i < a.length; i++)
18                if ( !Arrays.equals ( a[i], b[i]) ) {
19                    eq = false;
20                    break;
21                }
22        }
23        System.out.println("correct: identical = " + eq );
24    } }
```

Results of the comparisons:

- **K4B17E_ArrayEq_1_dim:**

The arrays **A** and **B** have the same content, but different references!

In corresponding array elements, the same values are stored (here: 0).

- **K4B18E_ArrayEq_2_dim:**

The arrays **A** and **B** do not have the same content (references!), but the sub-arrays have the same contents

(3) K4B19E_ArrayTests:

output method for (small) two-dimensional arrays:

```
1  ...
2
3  static String atos (int[][] array, String arrayName) {
4      String output = arrayName + ":\n";
5      for (int i = 0; i < array.length; i++) {
6          output += "[";
7          for (int j = 0; j < array[i].length; j++) {
8              output += "_" + array[i][j] + "_";
9          }
10         output += "]\n";
11     }
12     return output;
13 }
14
15 ...
```

(4) K4B19E_ArrayTests:

first attempt to copy a two-dimensional array:

```
1  ...
2
3      int [] [] a = { {22, 45, 57, 33},
4                      {64, 28, 19},
5                      {},
6                      {97},
7                      {88, 73, 44, 35, 84} };
8
9      int [] [] b = new int [5][];
10
11      for (int i = 0; i < a.length; i++)
12          b [i] = a [i];
13
14      a[1][1] = 0;
15
16      System.out.println( atos(a, "array a") );
17      System.out.println( atos(b, "array b") );
18
19  ...
```

(5) K4B19E_ArrayTests:

second attempt to copy a two-dimensional array:

```
1  ...
2
3      int [] [] c = { {22, 45, 57, 33},
4                      {64, 28, 19},
5                      {},
6                      {97},
7                      {88, 73, 44, 35, 84} };
8
9      int [] [] d = new int [5][];
10
11      for (int i = 0; i < c.length; i++)
12          d[i] = (int[]) c[i].clone();
13
14      c[1][1] = 0;
15
16      System.out.println( atos(c, "array c"));
17      System.out.println( atos(d, "array d"));
18
19  ...
```

results of the copy methods:

```
1 array a:  
2 [ 22  45  57  33 ]  
3 [ 64  0  19 ]  
4 []  
5 [ 97 ]  
6 [ 88  73  44  35  84 ]
```

```
1 array b:  
2 [ 22  45  57  33 ]  
3 [ 64  0  19 ]  
4 []  
5 [ 97 ]  
6 [ 88  73  44  35  84 ]
```

```
1 array c:  
2 [ 22  45  57  33 ]  
3 [ 64  0  19 ]  
4 []  
5 [ 97 ]  
6 [ 88  73  44  35  84 ]
```

```
1 array d:  
2 [ 22  45  57  33 ]  
3 [ 64  28  19 ]  
4 []  
5 [ 97 ]  
6 [ 88  73  44  35  84 ]
```

4 Non-primitive data types in Java

- Strings
- Arrays
- **StringTokenizer**
- StringBuffer

java.util.StringTokenizer

- allows to access the individual tokens ((e.g., words or numbers) in a String.
- constructor methods

```
1 public StringTokenizer (String str)
2
3 String st = "This is a String";
4 StringTokenizer tk1 = new StringTokenizer (st);
5 // breaks String down into tokens "This", "is", "a", and "String"
6 //-----
7 public StringTokenizer (String str, String delim)
8
9 String st2 = "17,_25,_77;_34_19";
10 StringTokenizer tk2 = new StringTokenizer (st2, "_;");
11 // breaks String down in tokens "17", "25", "77", "34", and "19",
12 // where " ", ",", and ";" are recognized as separation characters
13 //-----
14 public StringTokenizer (String str, String delim,
15                        boolean returnDelims)
16
17 StringTokenizer tk3 = new StringTokenizer (st2, "_;", true);
18 // additionally returns separation characters as tokens
```


methods:

```
1 public int countTokens ( )
2
3 int num = tk1.countTokens ( );
4 // yields the number of tokens in tk1
5 //-----
6
7 public boolean hasMoreTokens ( )
8
9 while ( tk1.hasMoreTokens ( ) ) { ... }
10 // checks if there are more tokens in tk1
11 //-----
12
13 public String nextToken ( )
14
15 String str;
16 str = tk1.nextToken ( );
17 // yields next token from tk1 and removes it
```

Example

```
1 import javax.swing.JOptionPane;
2 import java.util.*;
3
4 /* computes the sum of the input numbers */
5 public class K4B20E_TokenSum {
6     public static void main(String[] args) {
7         int sum = 0;
8         String numbers = JOptionPane.showInputDialog
9             ("input numbers," + "\n(separated by spaces): ");
10
11         StringTokenizer tokens = new StringTokenizer (numbers);
12
13         while (tokens.hasMoreTokens () )
14             sum = sum + Integer.parseInt (tokens.nextToken () );
15
16         JOptionPane.showMessageDialog (null, "Sum: " + sum );
17     }
18 }
```

4

Non-primitive data types in Java

- Strings
- Arrays
- StringTokenizer
- **StringBuffer**

`java.lang.StringBuffer` as supplement to `String`:

- An object of the class `String` is immutable, every manipulation of a `String` creates a new object.
- Objects of the class `StringBuffer` are mutable (can be modified).
- `String` is efficient when mostly static `Strings` are processed.
- `StringBuffer` is more efficient with highly dynamic `Strings`.

```

1 public class K4B21E_StringBufferConstructor {
2     public static void main( String args[] ) {
3
4         StringBuffer buffer1 = new StringBuffer();
5         StringBuffer buffer2 = new StringBuffer( 10 );
6         StringBuffer buffer3 = new StringBuffer( "hello" );
7
8         System.out.println(
9             "buffer1_=_\" + buffer1.toString() + "\"
10            + "\nbuffer2_=_\" + buffer2.toString() + "\"
11            + "\nbuffer3_=_\" + buffer3.toString() + "\"");
12     }
13 }

```

- line 4: empty StringBuffer with default capacity of 16 characters
- line 5: empty String buffer with capacity of 10 characters
- line 6: StringBuffer with capacity of 21 characters
(Contents: **hello**, additional space for 16 characters)
- Method **toString** yields String representation of the StringBuffer

Capacity of the buffer compared to the length of the String:

```
1 public class K4B22E_StringBufferCapacity {
2     public static void main( String args[] ){
3
4         StringBuffer buffer = new StringBuffer( "testing_a_buffer" );
5
6         System.out.println
7             ("buffer_=" + buffer.toString()
8              + "\nlength_=" + buffer.length()
9              + "\ncapacity_=" + buffer.capacity() );
10
11         buffer.ensureCapacity( 50 );
12         System.out.println
13             ("\nnew_capacity_=" + buffer.capacity());
14
15         buffer.setLength( 7 );
16         System.out.println
17             ("\nnew_length_=" + buffer.length()
18              + "\nbuffer=" + buffer.toString()
19              + "\ncapacity_=" + buffer.capacity());
20     }
21 }
```

- `StringBuffer("str")`: buffer length: `str + 16`;
- `ensureCapacity(n)`: buffer length: `max(n, 2*old value+2)`

With **public StringBuffer append(...)** Strings (and values of elementary data types) can be added to a StringBuffer:

```
1 public class K4B23E_StringBufferAppend {
2     public static void main( String args[] ){
3         String str = "7_";
4         int intVal = 4;
5         double floatVal = 11.0f;
6         boolean boolVal = true;
7         StringBuffer aBuffer = new StringBuffer( ",_that's_" );
8
9         StringBuffer buffer = new StringBuffer();
10
11        buffer.append(str);
12        buffer.append("+_");
13        buffer.append(intVal).append("_=");
14        System.out.println( buffer.toString()
15                               + "\ncapacity:_" + buffer.capacity() );
16
17        buffer.append(floatVal).append(aBuffer).append(boolVal);
18        System.out.println( buffer.toString()
19                               + "\ncapacity:_" + buffer.capacity() );
20    } }
```

append modifies the buffer AND returns a reference to it!

StringBuffers are used automatically to implement concatenation of Strings:

```
1  int one = 1;
2  String aString = "testing_...", anotherString;
3
4  anotherString = one + ",_2,_3,_" + aString;
```

is internally evaluated as follows:

```
1  anotherString = new StringBuffer()
2      .append(one)
3      .append(",_2,_3,_.")
4      .append(aString)
5      .toString();
```


More methods (also see the J2SE Documentation)

```
1 public StringBuffer insert (int offset, String str)
2 // inserts a String at position offset;
3 // overloaded method (also for int, long, char, ...)
4
5 public StringBuffer reverse ()
6 // reverse the contents of the buffer
7
8 public StringBuffer delete (int start, int end)
9 public StringBuffer deleteCharAt (int index)
10 // deletes a character sequence or a single character
11
12 public StringBuffer replace (int start, int end, String str)
13 // replaces a character sequence by str
14
15 public void setCharAt (int index, char ch)
16 // replaces a character at the given position
```

java.lang.Character includes some more methods for String processing (self-explaining...):

```
1 public static boolean isLetter (char ch)
2 public static boolean isDigit (char ch)
3 public static boolean isLetterOrDigit (char ch)
4 public static boolean isLowerCase (char ch)
5 public static boolean isUpperCase (char ch)
6 public static boolean isSpaceChar (char ch)
7 public static boolean isWhiteSpace (char ch)
8
9 public static String toString (char ch)
10
11 public static char toLowerCase (char ch)
12 public static char toUpperCase (char ch)
```

... and finally a (more clever) palindrome tester

```
1 public class K4B24E_Palindrom2 {
2
3 // remove white spaces, punctuation, etc.
4 static String removeJunk(String str) {
5     int len = str.length();
6     StringBuffer dest = new StringBuffer(len);
7     char c;
8     for (int i = 0; i < str.length(); i++) {
9         c = str.charAt(i);
10        if (Character.isLetterOrDigit(c)) {
11            dest.append(c);
12        }
13    }
14    return dest.toString();
15 }
16
17 // generate a reversed String
18 static String reverse(String string) {
19     StringBuffer sb = new StringBuffer(string);
20     return sb.reverse().toString();
21 }
22 ...
```

```
23 ...
24 // Test if palindrome (ignoring upper and lower case)
25 static boolean isPalindrome(String stringToTest) {
26     String workingCopy = removeJunk(stringToTest);
27     String reversedCopy = reverse(workingCopy);
28     return reversedCopy.equalsIgnoreCase(workingCopy);
29 }
30
31 public static void main(String[] args) {
32     String testString = System.console().readLine
33         ("input test string: ");
34     System.out.println("palindrome: " +
35         isPalindrome(testString) );
36 }
37 }
```

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

in the past:

- Programming languages have a fixed set of data types with predefined operations.

today:

- Since new application domains are identified all the time, it must be possible to add new data types and the corresponding operations.
- In addition the engineering of big systems must be better supported.

Class

- blueprint/template for the construction of new objects.

Object

- Representation of an element of a specific class with the operations that can be executed on it

Classes and objects support:

- **Abstraction**
provide an interface & hide the actual implementation.
- **Data encapsulation**
access to the data only through predefined methods.
- **Complex structures**
construction of complex data structures and their operators.
- **Reusability**
provide all-purpose program components.

An object has

- a **state** (defined by the values of its variables)
- a **behavior** (defined by its methods)

Example:

- A coin shows on its upside either “heads” or “tails”
- The upside can be changed, for example by a “coin toss”.
- The “state” of the object *coin* is its current upside (heads or tails).
- The “behavior” of the coin is that the coin can be tossed.
- The “behavior” of the coin can change its state.

A class is

- a blueprint for an object,
objects are generated using it as a template.

Example: **String**

- The class **String** is used to generate **String** objects.
- Every **String** object contains a sequence of characters defining its state.
- On every **String** object a set of methods can be applied.
- These methods offer services (e.g., **toUpperCase()**, **equals()** etc.), i.e., they define the behavior of a **String** object.
- Here, the behavior does not change the state of the object, but yields information and values for new objects.

First example, directory K5B01E_HeadsOrTails:

Java class for “coin tosses”, file CoinToss.java

```
1 public class CoinToss {
2     private final int HEADS = 1;
3     private final int TAILS = 0;
4     private int upside;
5
6     public CoinToss() {toss();}
7
8     public void toss() {upside = (int) (Math.random() * 2);}
9
10    public boolean isHeads() {return (upside == HEADS);}
11
12    @Override public String toString() {
13        String top;
14        if (upside == HEADS) top = "HEADS";
15        else top = "TAILS";
16        return top;
17    }
18 }
```

line 1	class header
lines 2-4	instance variables

lines 6	constructor
lines 8-17	methods for objects

Exampe: application of the 'coin tosses', HeadsOrTails.java

```
1 public class HeadsOrTails {
2     public static void main (String[] args) {
3         final int NUMBER_THROWS = 1000;
4         int heads = 0, tails = 0;
5
6         CoinToss myCoin = new CoinToss();
7
8         for (int count=1; count <= NUMBER_THROWS; count++) {
9             myCoin.toss();
10            if (myCoin.isHeads())
11                heads++;
12            else
13                tails++;
14        }
15
16        System.out.println(
17            "In " + NUMBER_THROWS + "_throws we had "
18            + heads + "_times heads and "
19            + tails + "_times tails.");
20    }
21 }
```

- Compilation, e.g., with
`javac HeadsOrTails.java`
- Then `HeadsOrTails.java` must have access to the file `CoinToss.class`. If this cannot be found, the compiler tries to generate it using
`javac CoinToss.java`
In this case, `CoinToss.java` must be available.
- Simplest solution: all files in the same directory.
- For the assignments you best create a new directory for every task!
- In big projects: use compiler option `-classpath` to define a search path. (→ `man java`)
- In `moodle` the files of a task are stored together in a directory.

Classes and Objects

- Introduction
- **Data encapsulation**
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Encapsulation:

- Objects can be seen from two viewpoints: internally and externally.
- From an **internal viewpoint**, an object is a collection of variables and methods accessing these variables.

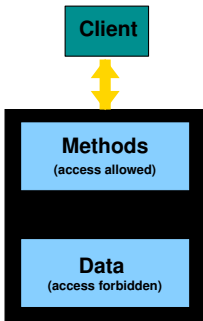
(This is not fully exact as methods belonging to objects of a class are stored only once for every class, not with every object)

- From an **external viewpoint**, an object is an encapsulated unit offering some services.

These services define the interface of the object.

- An object is thus an abstraction that hides details of the implementation from the rest of the system.
- An object can interact with other objects by using their services (i.e., by calling their methods).

- An (encapsulated) object can be seen as a black box.
- The inner details remain hidden to the caller of a method.
- An object should be 'self-controlled', i.e., every change of the object's state (the variables) should be done only using the methods provided by the object.



Constructors:

- special methods only used for generating a new object.
- are mostly used to initialize the variables of an object.
- always have the same name as the class.
- can be overloaded.
- can have parameters, but do not return anything.

If not specified, a default constructor is used (corresponding to a constructor with empty parameter list and empty method body).

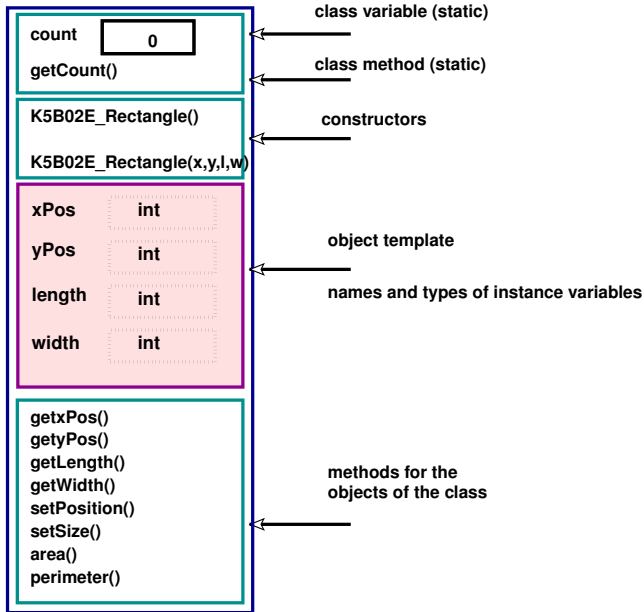
Example K5B02E_Rectangle: rectangles in a plane

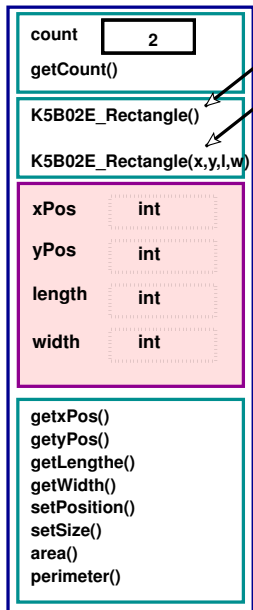
```
1 public class Rectangle {
2
3     static int count; class variable (static)
4
5     private int xPos, yPos, width, height; instance variables
6
7     Rectangle () { count++; } constructor 1
8
9     Rectangle (int x, int y, int w, int h) { constructor 2
10         setPosition (x, y); setSize (w, h); count++; }
11
12     static int getCount () {return count;} class method (static)
13
14     int getXPos ()    { return xPos;} object methods
15     int getYPos ()    { return yPos;}
16     int getHeight ()  { return height;}
17     int getWidth ()  { return width;}
18
19     void setPosition (int x, int y) { xPos = x; yPos = y; }
20     void setSize (int w, int h) { width = w; height = h; }
21
22     int area () { return width * height; }
23     int perimeter () { return 2 * ( width + height ); }
24 }
```

```

1 public class RectangleTest {
2     public static void main(String[] args) {
3
4         Rectangle r1 = new Rectangle ();
5         r1.setSize (4, 12);
6
7         Rectangle r2  = new Rectangle (3, 5, 12, 19);
8
9         System.out.println(
10            "Rectangle r1:"
11 + "\nx_=====" + r1.getxPos()    + ",_y_=====" + r1.getyPos()
12 + "\nWidth= " + r1.getWidth() + ",_height  = " + r1.getHeight()
13 + "\nArea= " + r1.area()    + ",_perimeter= " + r1.perimeter()
14 + "\n\n" +
15            "Rectangle r2:"
16 + "\nx_=====" + r2.getxPos()    + ",_y_=====" + r2.getyPos()
17 + "\nWidth= " + r2.getWidth() + ",_height  = " + r2.getHeight()
18 + "\nArea= " + r2.area()    + ",_perimeter= " + r2.perimeter()
19 + "\n\n"+
20            "generated objects: " + Rectangle.getCount ());
21     }
22 }

```

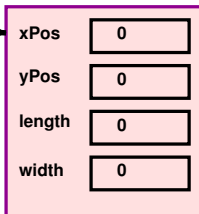




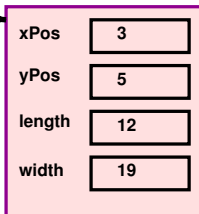
```
K5B02E_Rectangle r1 = new K5B02E_Rectangle();  
K5B02E_Rectangle r2 = new K5B02E_Rectangle(3,5,12,19);
```



r1



r2



Visibility modifiers

- In Java encapsulation is achieved using visibility modifiers.
- A *modifier* is a reserved word that determines several characteristics (of methods or variables).
If for example a data element is defined with the modifier **final**, this is treated as a constant.
- Java has four options for the visibility of elements of a class (i.e., for methods, variables, constants):
 - ▶ **public**: can be used from everywhere.
 - ▶ **private**: can be used only from within the class.
 - ▶ **protected**: see later (inheritance)
 - ▶ *without modifier* ('package private'): can be used only within the same 'package'.
- **public** variables should be avoided, usually every variable should be 'private'.

Visibility modifiers for **methods**:

- Methods that offer services included in the interface of the object are usually defined as '**public**'.
- Methods that are only used by other methods of the class ('support methods') should be declared '**private**'.

	public	private
variables	violation of the encapsulation	support of the encapsulation
methods	service for 'clients'	support for other methods of the class

visibility modifiers for **classes**:

- **public**:
visible also outside the own 'package'.
- *without modifier* ('package private'):
visible only in the same 'package'.

static: important modifier for **class or object scope** of methods and variables:

- **static**:

A '**static**' method can be called without the existence of an object (e.g., `Rectangle.getCount()`).

A '**static**' variable is stored in the memory area of the class (e.g., '**count**' in the 'Rectangle' example).

With other classes the class name must be used, (e.g., `Rectangle.getCount()` or `Rectangle.count`).

- not **static** (default!):

Non-**static** variables are stored in the memory area of an object.

Non-**static** variables and methods thus always refer to a specific object (e.g., `r1.getWidth()`)

setter and getter methods:

- By the principle of encapsulation, a variable should not be directly accessible from the outside.
- Instead:
 - ▶ Variables should be declared as **private**, in addition optional
 - ▶ **public getter** method for reading its value and
 - ▶ **public setter** method for modifying its value

for example:

```
1 ...  
2 private int value;  
3 public int getValue () {  
4 ...  
5 }  
6 public void setValue (int newValue) {  
7 ...  
8 }
```


Scope of variables:

- The variables and methods defined at class level are also called class members.
- Every class method can access all class members, i.e., all class members are valid in every class method.
- The variables defined at method level are only known in the local method.
- Local variables can hide class variables.

Referencing class members:

- Members are referenced by their name...
- in the *local* object directly by variable or method name
- when hidden by a local variable referenced through **this** in the following form
this.Name
- in a *different* object or a *different* class:
ReferenceName.Name or even
ReferenceName1.ReferenceName2.Name

- The keyword **this** allows objects to reference themselves
- This can be used to access instance variables hidden by local variables:

```
1 public class Rectangle {  
2     static int count;  
3     private int xPos, yPos, width, height ;  
4     Rectangle () { count++; }  
5     Rectangle (int x, int y, int w, int h) {  
6         setPosition (x, y); setSize (w, h); count++; }  
7  
8     ...  
9  
10    void setSize (int width, int height) {  
11        this.width = width; this.height = height;    }  
12  
13    int area () { return width * height; }  
14    int perimeter () { return 2 * ( width + height ); }  
15 }
```

Classes and Objects

- Introduction
- Data encapsulation
- **Example: Rational Numbers**
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

In the following:

- class for representing and manipulating rational numbers
- every object of this class represents a rational number
- storage as pair (numerator, denominator).
- with methods for basic arithmetics, test for equality, conversion to Strings

```
1 public class RationalNumber {
2     private int numerator, denominator;
3
4     //-----
5     //  Constructor:
6     //  - Initialization of an object "rational number"
7     //  - parameter values assigned to variables
8     //  - numerator contains the sign
9     //  - representation in canonical form (cancel)
10    //  - no check if denominator is 0
11    //-----
12    public RationalNumber (int num, int denom) {
13
14        if (denom < 0) {
15            num = -num;
16            denom = -denom;
17        }
18        numerator = num;
19        denominator = denom;
20
21        cancel();
22    }
```

```
23 //-----
24 //  retrieves numerator
25 //-----
26 public int getNumerator () {
27     return numerator;
28 }
29
30 //-----
31 //  retrieves denominator
32 //-----
33 public int getDenominator () {
34     return denominator;
35 }
36
37 //-----
38 //  yields the inverse as a rational number
39 //-----
40 public RationalNumber inverse () {
41     return new RationalNumber (denominator, numerator);
42 }
```

```
43 //-----
44 //  - addition of two rational numbers
45 //  - returns the sum as rational number
46 //-----
47 public RationalNumber add (RationalNumber op2) {
48     int commonDenominator = denominator * op2.getDenominator();
49     int numerator1 = numerator * op2.getDenominator();
50     int numerator2 = op2.getNumerator() * denominator;
51     int sum = numerator1 + numerator2;
52     return new RationalNumber (sum, commonDenominator);
53 }
54
55 //-----
56 //  - subtraction (this number - parameter op2)
57 //  - returns difference
58 //-----
59 public RationalNumber subtract (RationalNumber op2) {
60     int commonDenominator = denominator * op2.getDenominator();
61     int numerator1 = numerator * op2.getDenominator();
62     int numerator2 = op2.getNominator() * denominator;
63     int difference = numerator1 - numerator2;
64     return new RationalNumber (difference, commonDenominator);
65 }
```



```
66 //-----
67 // - multiplication of two rational numbers
68 // - returns product as rational number
69 //-----
70 public RationalNumber multiply (RationalNumber op2) {
71     int num = numerator * op2.getNumerator();
72     int denom = denominator * op2.getDenominator();
73     return new RationalNumber (num, denom);
74 }
75
76 //-----
77 // - division (this number / parameter op2)
78 // - returns quotient as rational number
79 //-----
80 public RationalNumber divide (RationalNumber op2) {
81     return multiply (op2.inverse());
82 }
```

```

83      //-----
84      //   compare two rational numbers
85  // (in canonical form by construction)
86      //-----
87  public boolean equals (RationalNumber op2) {
88      return ( numerator == op2.getNumerator()
89              && denominator == op2.getDenominator() );
90  }
91
92      //-----
93      //   transform a rational number to a String
94      //-----
95  @Override public String toString () {
96      String result;
97      if (numerator == 0)
98          result = "0";
99      else
100          if (denominator == 1)
101              result = numerator + "";
102          else
103              result = numerator + "/" + denominator;
104      return result;
105  }

```

```

105 //-----
106 //  cancel a rational number
107 // (i.e., convert it into its canonical form)
108 //-----
109 private void cancel () {
110     if (numerator != 0) {
111         int common = gcd (Math.abs(numerator), denominator);
112         numerator = numerator / common;
113         denominator = denominator / common;
114     }
115 }
116
117 //-----
118 //  - greatest common divisor of two integers
119 //  - returns gcd
120 //-----
121 private int gcd (int number1, int number2) {
122     while (number1 != number2)
123         if (number1 > number2)
124             number1 = number1 - number2;
125         else
126             number2 = number2 - number1;
127     return number1;
128 }
129 }

```

```
1 import java.util.Scanner;
2 import java.util.*;
3
4 public class RationalNumbers {
5     public static void main (String[] args) {
6         Scanner sc = new Scanner(System.in);
7
8         RationalNumber r1, r2;
9
10        int numerator, denominator;
11
12        char operator;
13
14        System.out.println("Input:\n" +
15            "(Syntax: _number|number [+*/] number|number [+*/]... )" );
16
17        String input = sc.nextLine();
18
19        StringTokenizer tokens =
20            new StringTokenizer (input, "|_", true);
```

```
21     numerator = Integer.parseInt (tokens.nextToken () );
22     tokens.nextToken();
23     denominator = Integer.parseInt (tokens.nextToken () );
24
25     r1 = new RationalNumber (numerator, denominator);
26
27     while (tokens.hasMoreTokens () ) {
28         operator = tokens.nextToken().charAt(0);
29
30         numerator = Integer.parseInt (tokens.nextToken () );
31         tokens.nextToken();
32         denominator = Integer.parseInt (tokens.nextToken () );
33
34         r2 = new RationalNumber (numerator, denominator);
35
36         switch (operator) {
37             case '+': r1 = r1.add (r2);           break;
38             case '-': r1 = r1.subtract (r2);     break;
39             case '*': r1 = r1.multiply (r2); break;
40             case '/': r1 = r1.divide (r2);       break;
41         }
42     }
43     System.out.println ("result: " + r1.toString() );
44 }
45 }
```

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- **Wrapper classes**
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

'wrapper' classes ('Boxing'):

In Java `int`, `long`, `short`, `byte`, `char`, `float`, `double` and `boolean` are primitive data types and not classes. Thus such types cannot be used where object references are expected.

Sometimes it can be useful to treat primitive values as objects. Java provides special classes ('wrapper') that allow to 'wrap' a value into an object:

primitive type	wrapper class
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>char</code>	<code>Character</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

Example:

long m



Long n



Example: `java.lang.Integer`

```
1 public final class Integer
2         extends Number implements Comparable<Integer>
```

- The class `Integer` wraps an `int` value into an object.
- `Integer` objects contain a single `int` data field.

Constructors:

- `Integer (int value)`, i.e. `int`-Wert \leadsto `Integer` object ('wrap')
- `Integer (String s)`, i.e. `String` \leadsto `Integer` object

Constants (selection):

- `static int MAX_VALUE (= $2^{31} - 1$);`
- `static int MIN_VALUE (= -2^{31});`

Small selection of the methods of **public final class** Integer :

```
1 public static int parseInt(String s)
2     // generates int value from String
3
4 public static Integer valueOf(String s)
5     // generates Integer object from String
6
7 public int intValue ()
8     //generates int value from Integer object
9     //( 'unwrap' )
10
11 public int compareTo(Integer iObj)
12     //numeric comparison of two Integer objects
13     // e.g.: x = int1.compareTo(int2);
14     //      x == 0 if: intValue(int1) == intValue(int2)
15     //      x < 0  if: intValue(int1) <  intValue(int2)
16     //      x > 0  if: intValue(int1) >  intValue(int2)
17
18 public static String toString(int i)
19     // generates String from int value
20
21 public String toString()
22     // generates String from Integer object
```

Auto-Boxing, Auto-Unboxing: Automatic conversion between wrapper class and primitive type (auto-boxing, auto-unboxing)

- explicit conversion (required until Java 1.4.2):

```
1 int i = 15;
2 Integer iObj = new Integer (i);
3 i = iObj.intValue();
4
5 Integer [] iArray = new Integer [10];
6 iArray [5] = new Integer (54286);
7 int value = iArray [5].intValue();
```

- Auto-(Un)boxing (since Java 1.5/5.0):

```
1 int i = 15;
2 Integer iObj = i;
3 i = iObj;
4
5 Integer [] iArray = new Integer [10];
6 iArray [5] = 54286;
7 int value = iArray [5];
```

Corresponding Example K5B04E_AutoBox:

```
1 public class AutoBox {
2     public static void main (String args[]) {
3
4         int i = 54286;
5
6         Integer iObj = new Integer (i);
7         Integer jObj = new Integer (i);
8
9         if (i == iObj)
10             System.out.println( "i_=_iObj_=_ " + iObj + "_(values!)\n");
11
12         if (jObj != iObj)
13             System.out.println( "jObj_!=_iObj_(references!)\n");
14
15         i = iObj / 2;
16         System.out.println( "iObj_/_2_=_ " + i + "_(Auto-Unboxing)");
17     }
18 }
```

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- **Standard datatypes Queue, Stack, List**
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Addresses, Pointer, References:

- Every data item stored in a computer has a (memory) **address**.
- A **pointer** is an address and is used to
 - ▶ locate and identify variables, Strings, arrays and other complex structures
 - ▶ construct more complex structures by chaining simple structures
- Some programming languages allow to compute with pointers ('pointer swizzling')
- **References** are pointers, but with restricted semantics: In Java references can only refer to existing objects, 'pointer swizzling' is not possible.

References can be used to combine simple elements to **dynamic data structures**.

Examples are

- **Queue**

Collection of elements, organized following the **FIFO** principle (first in, first out).

can be used when data or events should be processed in the order of their arrival.

- **Stack**

Collection of elements, organized following the **LIFO** principle (last in, first out).

can be used to for resolving recursion or for processing context-free languages

- **List**

Linear collection of elements, may be sorted.

Basic class 'Elem' for the construction of complex structures (List, Stack, Queue, ...):

```
1 public class Elem {  
2     public Elem () { } constructor  
3     public Elem (Object obj) { setObject(obj); } constructor  
4  
5     private Object obj; data (arbitrary object)  
6     private Elem next; reference to next element  
7  
8     public void setObject (Object newObj) { data access method  
9         obj = newObj;  
10    }  
11    public Object getObject () { data access method  
12        return obj;  
13    }  
14    public void setNext (Elem nextElem) { access to reference  
15        next = nextElem;  
16    }  
17    public Elem getNext () { access to reference  
18        return next;  
19    }  
20  
21    @Override public String toString () { return obj.toString (); }  
22 }
```

- Every instance `x` of `Elem` (i.e., every object of this class) includes as instance variables `Object obj` and `Elem next`.
- Here, `x` does not store the actual data of these objects, but only references to them.
- In particular, this can be the `null` reference, for which one must watch out, for example

```
y = x.getNext(); if (y != null ) {...process y...}.
```

- Self-reference is often intended, for example `x.setNext(x)`.
- `setObject` can take arbitrary *objects* even auto-boxing using wrapper classes is allowed, e.g., `x.setObject("text")` or `x.setObject(42)`.
- `getObject` returns an *object* that needs to be casted to the intended data type, e.g.,

```
String s = (String) x.getObject(),  
Integer i = (Integer) x.getObject() or even  
int i = (Integer) x.getObject() (via auto-unboxing).
```
- More on casts and `@Override` in the chapter about *inheritance*!

Queue: The data structure **Queue** can be characterized by two operations:

- **enqueue**: inserts an object at the end of the queue.
- **dequeue**: removes the first object from the queue.

more useful operations:

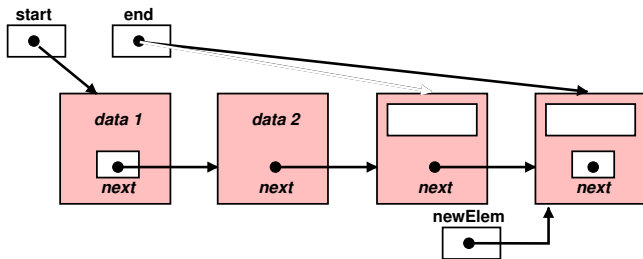
- **isEmpty**: checks if the queue is empty.
- **isFull**: checks if the queue is empty or if the maximal size is reached.
- **peek**: returns the first object, but does not remove it

Queue based on objects of the class **Elem**:

```
1 public class Queue {
2     public Queue() { }
3     private Elem start, end; chaining
4
5     public void enqueue (Elem newElem) { insert an element
6         if (start == null) start = newElem;
7         else end.setNext (newElem);
8         end = newElem;
9     }
10    public Elem dequeue () { remove the 'oldest' element
11        if (start == null) return null;
12        Elem temp = start;
13        start = start.getNext();
14        if (start == null) end = null;
15        return temp;
16    }
17    @Override public String toString () { ... for debugging ...
18        Elem position = start;
19        String str = "";
20        while (position != null) {
21            str += position.getObject().toString() + "  ";
22            position = position.getNext();
23        }
24        return str;
25    } }
```

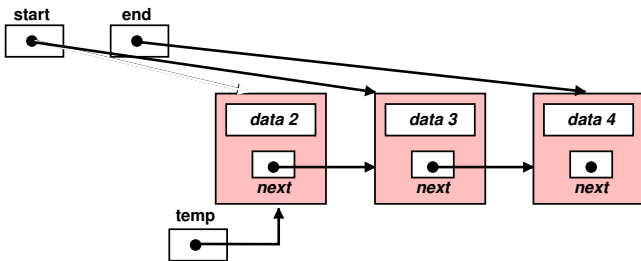
enqueue: Insert an element at the end of the queue

```
1  public void enqueue (Elem newElem) {  
2      if (start == null)  
3          start = newElem;  
4      else  
5          end.setNext (newElem);  
6      end = newElem;  
7  }  
8  
9  ...
```



deQueue: remove the first element

```
1 public Elem deQueue () {  
2     if (start == null)  
3         return null;  
4     Elem temp = start;  
5     start = start.getNext();  
6     if (start == null)  
7         end = null;  
8     return temp;  
9 }
```



Queue

with **visible** element structure

```
1 public class Queue {
2     public Queue() {}
3     private Elem start, end;
4
5     public void enqueue
6         (Elem newElem) {
7
8         if (start == null)
9             start = newElem;
10        else
11            end.setNext (newElem);
12        end = newElem;
13    }
14
15    public Elem dequeue () {
16        if (start == null)
17            return null;
18        Elem temp = start;
19        start = start.getNext();
20        if (start == null)
21            end = null;
22        return temp;
23    }
```

modified queue

with **hidden** element structure

```
1 public class Queue {
2     public Queue() {}
3     private Elem start, end;
4
5     public void enqueue
6         (Object newObj) {
7         Elem newElem =
8             new Elem(newObj);
9         if (start == null)
10            start = newElem;
11        else
12            end.setNext (newElem);
13        end = newElem;
14    }
15    public Object dequeue () {
16        if (start == null)
17            return null;
18        Elem temp = start;
19        start = start.getNext();
20        if (start == null)
21            end = null;
22        return temp.getObject();
23    }
```

interface vs. implementation:

A class and its objects provide a unique **interface**, defined by the methods and variables visible from the outside.

A class C_1 can be replaced by a class C_2 , if C_2

- provides a syntactically and semantically identical interface to the outside world
- uses only services of existing classes

Thus: the implementation of a class can be changed locally, if

- the interface is not changed
- no services of additional classes are needed

Examples K5B05E_... with two implementations of Queue:

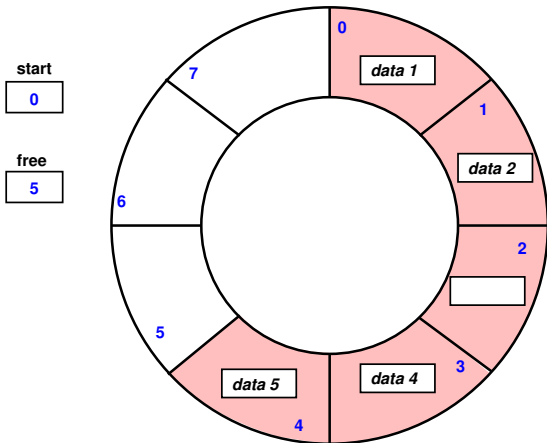
K5B05E_Queue_Linked
(with linked elements)

```
1 public class Queue {
2
3     public Queue () {}
4
5     private Elem start, end;
6
7
8
9
10
11
12     public void enqueue
13         (Object newObj) {
14         Elem newElem =
15             new Elem(newObj);
16         if (start == null)
17             start = newElem;
18         else
19             end.setNext (newElem);
20         end = newElem;
21     }
22     ...
```

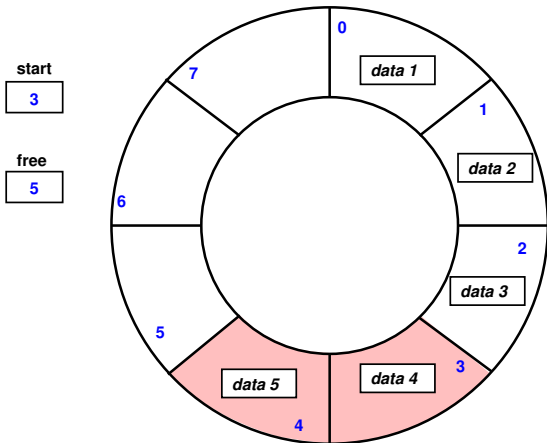
K5B05E_Queue_Array
(based on array)

```
1 public class Queue {
2
3     public Queue () {}
4
5     private int start, free,
6         length = 8;
7     private boolean empty = true,
8         full = false;
9     private Object array []
10         = new Object[length];
11
12     public void enqueue
13         (Object newObj) {
14         if ( !full ) {
15             array [free] = newObj;
16             free = (free+1) % length;
17             empty = false;
18             full = ( start == free );
19         }
20     }
21
22     ...
```

Queue based on array: **enQueue** operations:



Queue based on array: **deQueue** operation:



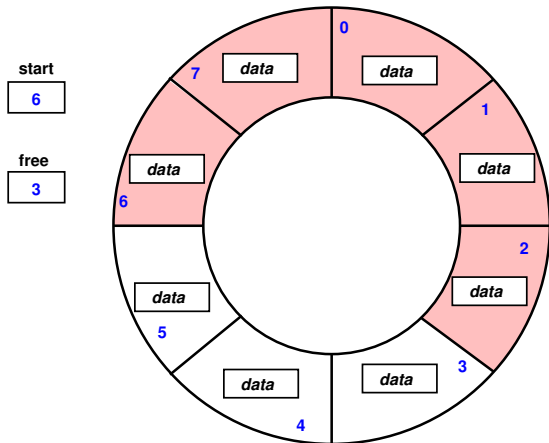
```
1  ...
2
3  public Object deQueue () {
4      if (start != null) {
5          Elem temp = start;
6          start = start.getNext();
7          if (start == null)
8              end = null;
9          return temp.getObject();
10     }
11     else return null;
12 }
13
14 public boolean isEmpty () {
15     return (start == null);
16 }
17
18 ...
```

```
1  ...
2
3  public Object deQueue () {
4      if ( !empty ) {
5          Object temp = array [start];
6          start = (start+1) % length;
7          full = false;
8          empty = ( start == free );
9          return temp;
10     }
11     else return null;
12 }
13
14 public boolean isEmpty () {
15     return empty;
16 }
17
18 ...
```

```
1  ...
2
3  public boolean isFull () {
4      return false;
5  }
6
7  @Override
8  public String toString () {
9      String str = "";
10     Elem position = start;
11     while (position != null) {
12         str += position.
13             getObject().
14             toString() + "_";
15         position = position.
16             getNext();
17     }
18     return str;
19 }
20
21 }
```

```
1  ...
2
3  public boolean isFull () {
4      return full;
5  }
6
7  @Override
8  public String toString () {
9      String str = "";
10     int i = start;
11     if (!empty)
12         do {
13             str += array [i] + "_";
14             i = (i + 1) % length;
15         }
16         while (i != free);
17
18     return str;
19 }
20
21 }
```

Queue based on array: **ring buffer technique**:



```

1 public class QueueTest {
2     public static void main(String[] args) {
3         Queue qu = new Queue ();
4         while ( true ) {
5             String str = System.console().readLine
6                 ("Input [ key|-|0]: ");
7             if ( str.equals("0") ) return;
8             if ( str.equals("-") ) {
9                 int key = (Integer)qu.dequeue();
10                System.out.println("deQueue:_ " + key);
11                System.out.println("Queue:___" + qu);
12            } else {
13                int key = Integer.parseInt(str);
14                qu.enqueue (key);
15                System.out.println("enQueue:_ " + key);
16                System.out.println("Queue:___" + qu);
17            }
18        }
19    }
20 }

```

In the examples K5B05E_Queue_Linked und K5B05E_Queue_Array the QueueTest.java files are identical! All differences are in the file corresponding to Queue.java.

The data structure **stack** can be characterized by two operations:

- **push:**
puts an object on the stack.
- **pop:**
removes the top-most object from the stack.

Other useful operations:

- **isEmpty:**
checks if the stack is empty.
- **isFull:**
checks if the stack is full or its capacity is reached.
- **peek:**
returns the top-most element of the stack without removing it.
(can be replaced by a sequence of pop and push)

Example K5B06E_Stack_... with two implementations of a stack:

K5B06E_Stack_Linked
(of linked elements)

```
1 public class Stack {
2
3     public Stack () {}
4
5     private Elem top;
6
7
8
9
10
11    public void push
12        (Object newObj) {
13        Elem newElem =
14            new Elem(newObj);
15        newElem.setNext(top);
16        top = newElem;
17    }
18
19    ...
```

K5B06E_Stack_Array
(based on array)

```
1 public class Stack {
2
3     public Stack () {}
4
5     private int free, length=50;
6     private boolean empty=true,
7                     full=false;
8     private Object array [] =
9         new Object [length];
10
11    public void push
12        (Object newObj) {
13        if ( !full ) {
14            array [free++] = newObj;
15            empty = false;
16            full = (free == length );
17        }
18    }
19    ...
```

```
1  ...
2
3  public Object pop () {
4      if (top != null) {
5          Elem temp = top;
6          top = top.getNext();
7          return temp.getObject();
8      }
9      else return null;
10 }
11
12
13
14 public boolean isEmpty () {
15     return (top == null);
16 }
17
18 ...
```

```
1  ...
2
3  public Object pop () {
4      if ( !empty ) {
5          free--;
6          Object temp = array[free];
7          full = false;
8          empty = ( free == 0 );
9          return temp;
10     } else return null;
11 }
12
13
14 public boolean isEmpty () {
15     return empty;
16 }
17
18 ...
```

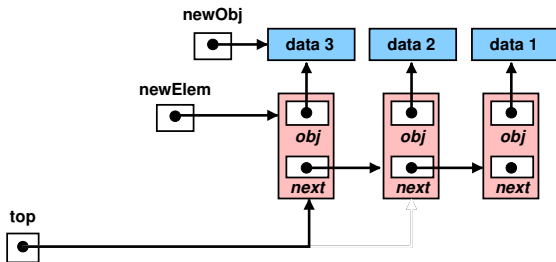


```
1  ...
2
3  public boolean isFull () {
4      return false;
5  }
6
7  @Override
8  public String toString () {
9      String str = "";
10     Elem position = top;
11
12     while (position != null) {
13         str += position.
14             getObject().
15             toString() + "_";
16         position = position.
17             getNext();
18     }
19
20     return str;
21 }
22 }
```

```
1  ...
2
3  public boolean isFull () {
4      return full;
5  }
6
7  @Override
8  public String toString () {
9
10     String str = "";
11
12     for (int i = free-1;
13         i >= 0;
14         i--) {
15         str += array[i] + "_";
16     }
17
18
19     return str;
20 }
21 }
22 }
```

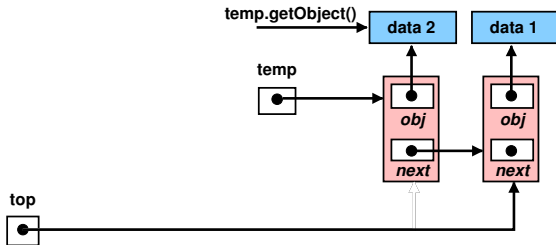
push: Put an element on the stack

```
1 public void push
2     (Object newObj) {
3     Elem newElem =
4         new Elem(newObj);
5     newElem.setNext(top);
6     top = newElem;
7 }
8     ...
```



pop: Take the top-most element from the stack

```
1 public Object pop () {  
2     if (top != null) {  
3         Elem temp = top;  
4         top = top.getNext();  
5         return temp.getObject();  
6     }  
7     else return null;  
8 }
```



```

1 public class StackTest {
2     public static void main(String[] args) {
3         Stack st = new Stack();
4         while ( true ) {
5             String str = System.console().readLine
6                 ("Input [key|-|0]: ");
7             if ( str.equals("0") ) return;
8             if ( str.equals("-") ) {
9                 int key = (Integer)st.pop();
10                System.out.println("pop:_" + key);
11                System.out.println("Stack:_" + st);
12            } else {
13                int key = Integer.parseInt(str);
14                st.push(key);
15                System.out.println("push:_" + key);
16                System.out.println("Stack:_" + st);
17            }
18        }
19    }
20 }

```

In both K5B06E_Stack_Linked and K5B06E_Stack_Array the file StackTest.java is identical! All differences are in the file Stack.java.

Lists:

- In contrast to Queue and Stack, a **List** allows to insert and delete at arbitrary positions.
- Variant we consider here: '**sorted List**', for simplicity based on Integer elements.
- this requires a sorting key, for example:

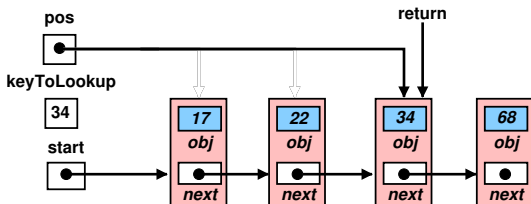
```
1 public static int getKey(Object obj) { // sorting key
2     return (Integer)obj; // only for Integer objects
3 }
```

```
1 public class SortedList {
2
3     public SortedList ()    { }    Constructor
4
5     private Elem start;    anchor of the list
6
7     methods for list accesses:
8     public static int getKey(Object obj)    { ... }
9     public Elem lookUp (int keyToLookup)    { ... }
10    public void sortIn (Elem newElem)    { ... }
11    public void delete (Elem elemToDelete) { ... }
12
13    @Override public String toString () { ...for debugging...
14        String str = "";
15        for (Elem pos = start; pos != null; pos = pos.getNext())
16            str += pos.getValue() + "_";
17        return str;
18    }
19 }
```

```

1  public Elem lookUp (int keyToLookup)  {
2      Elem pos = start;
3      while ( pos != null ){
4          if ( getKey(pos.getObject()) == keyToLookup )
5              return pos;                // found
6          else
7              pos = pos.getNext();
8      }
9      return null;                        // not found
10 }

```



Attention: in this and the following slides only the key of the data object is shown (instead of a reference and the actual object).

Basic structure of sortIn:

```
1  public void sortIn (Object newObj) {
2      int key = getKey(newObj);
3      Elem newElem = new Elem(newObj);
4      // insert into empty list
5      if (start == null ) {
6          ...
7      }
8      // insert before first element:
9      if (getKey(start.getObject()) > key ) {
10         ...
11     }
12     // insert between two elements:
13     Elem pos = start;
14     while ( pos.getNext() != null)
15         if (getKey(pos.getNext().getObject()) > key)
16             ... else ...
17     // insert at end of list:
18     pos.setNext(newElem);
19     newElem.setNext(null);
20 }
```

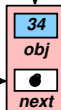

Insert into empty list:

```
1  int key = getKey(newObj);  
2  Elem newElem = new Elem(newObj);  
3  
4  if (start == null ) {  
5      newElem.setNext (null);      start = newElem;  
6      return;  
7  }
```

newElem

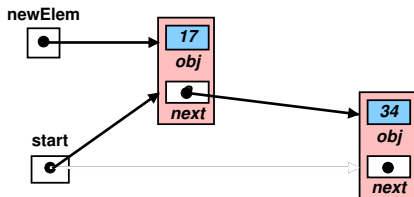


start



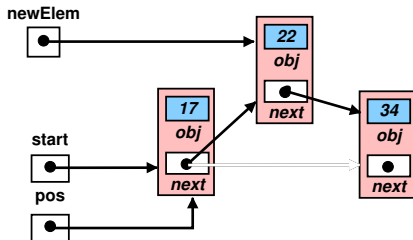
Insert before first element:

```
1  if (getKey(start.getObject()) > key ) {  
2      newElem.setNext (start);    start = newElem;  
3      return;  
4  }
```



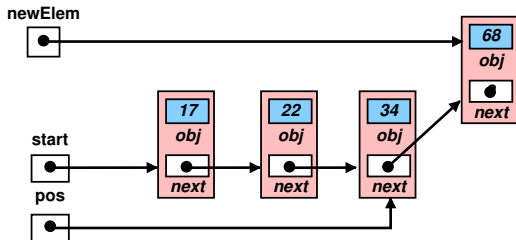
Insert between two elements:

```
1 Elem pos = start;
2 while ( pos.getNext() != null)
3     if (getKey(pos.getNext().getObject()) > key) {
4         newElem.setNext (pos.getNext());
5         pos.setNext (newElem);
6         return;
7     } else
8         pos = pos.getNext();
```



Insert at end of list: (`pos` is already set!)

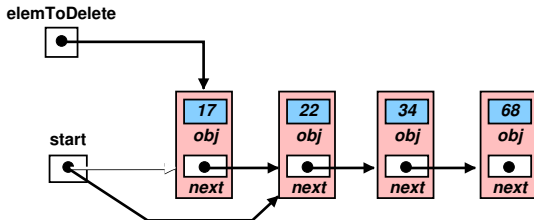
```
1 pos.setNext(newElem);  
2 newElem.setNext(null);
```



```
1  public void delete (Elem elemToDelete) {
2  // empty list or nothing to delete
3      if (start == null || elemToDelete == null)
4          return;
5  // delete the first element
6      if (start == elemToDelete) {
7          start = elemToDelete.getNext();
8          return;
9      }
10 // delete element at other position
11     Elem pos = start;
12     while ( pos.getNext() != null )
13         if ( pos.getNext() == elemToDelete ) {
14             pos.setNext(elemToDelete.getNext());
15             return;
16         }
17     else
18         pos = pos.getNext();
19 }
```

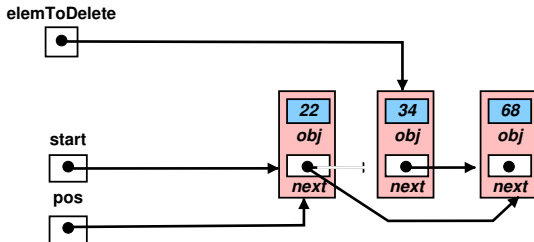
Delete first element:

```
1 if (start == elemToDelete) {  
2     start = elemToDelete.getNext();  
3     return;  
4 }
```



Delete other elements:

```
1  Elem pos = start;  
2  while ( pos.getNext() != null )  
3      if ( pos.getNext() == elemToDelete ) {  
4          pos.setNext( elemToDelete.getNext() );  
5          return;  
6      }  
7  else  
8      pos = pos.getNext();
```



Test Program for SortedList in K5B07E_SortedList:

```
1 public class SortedListTest {
2     public static void main(String[] args) {
3         SortedList liste = new SortedList();
4         while ( true ) {
5             String str = System.console().readLine
6                 ("Input [key|-key|0]: ");
7             if (str.equals("0")) {
8                 return;
9             } else {
10                int key = Integer.parseInt(str);
11                if (key > 0) {
12                    liste.sortIn (new Integer(key));
13                    System.out.println("Insert:_" + key);
14                    System.out.println("List:___" + liste);
15                } else {
16                    key = - key;
17                    liste.delete (liste.lookup (key));
18                    System.out.println("Delete:_" + key);
19                    System.out.println("List:___" + liste);
20                }
21            }
22        }
23    }
24 }
```


Doubly linked lists:

- A list is processed following the references.
- With a singly linked list, a step backwards usually requires to start again at the head of the list.
- If backward steps are frequent, a **doubly linked list** should be preferred.

To demonstrate the use of references, we discuss a simplified version (e.g., without special methods for backward search).

In the example `K5B08E_SortedDoubleList` the test class `SortedListTest` is identical to the previous example, but the classes `Elem` and `SortedList` are modified.

Element with forward and backward linkage, with additional `prev` reference

```
1 public class DLElem {
2
3     private Object obj;
4     private DLElem next;
5     private DLElem prev;           reference to previous element
6
7     public DLElem () {}
8     public DLElem (Object obj) { setObject(obj); }
9
10    public void setObject (Object newObj) { obj = newObj; }
11    public Object getObject () { return obj; }
12
13    public void setNext (DLElem nextElem) { next = nextElem; }
14    public void setPrev (DLElem prevElem) { prev = prevElem; }
15
16    public DLElem getNext () { return next; }
17    public DLElem getPrev () { return prev; }
18
19    @Override
20    public String toString () { return obj.toString (); }
21 }
```

```

1 public class SortedDoubleList {
2
3     public SortedDoubleList () { }      constructor
4     private DLElem start;                start of list
5     private DLElem end;                  end of list
6     sorting key as before:
7     public static int getKey(Object obj) { // sorting key
8         return (Integer)obj;           // only for Integer objects
9     }
10    identical to single linkage:
11    public DLElem lookUp (int keyToLookup) { ... }
12    more methods for list access
13    public void sortIn (Object newObj)      { ... }
14    public void delete delete (DLElem elemToDelete) { ... }
15
16    for debugging: list forward and backward...
17    @Override public String toString () {
18        String str = " _---> _";
19        for (DLElem pos = start; pos != null; pos = pos.getNext())
20            str += (Integer)pos.getObject() + " _ _";
21        str += ", _ _<--- _";
22        for (DLElem pos = end; pos != null; pos = pos.getPrev())
23            str += (Integer)pos.getObject() + " _ _";
24        return str;
25    } }

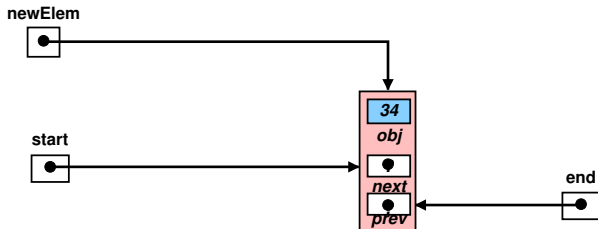
```

The other methods are significantly different from the previous example:

```
1 public void sortIn (Object newObj) {
2     int key = getKey(newObj);
3     DLElem newElem = new DLElem(newObj);
4     //insert in empty list
5     if (start == null ) {
6         ...
7     }
8     // insert before first element
9     if ( getKey(start.getObject()) > key ) {
10         ...; return;
11     }
12     // insert between two elements
13     DLElem pos = start;
14     while ( pos.getNext() != null)
15         if ( getKey(pos.getNext().getObject()) > key ) {
16             ...; return;
17         }
18     else pos = pos.getNext();
19     // insert at the end of the list
20     pos.setNext(newElem);    end = newElem;
21     newElem.setNext(null);   newElem.setPrev(pos);
22 }
```

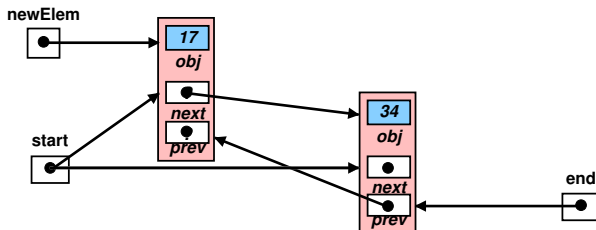
Insert in empty list:

```
1  if (start == null ) {  
2      newElem.setNext (null);  newElem.setPrev (null);  
3      start = newElem;  end = newElem;  
4      return;  
5  }
```



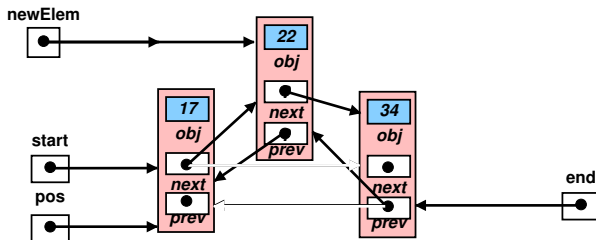
Insert before first element:

```
1  if ( getKey(start.getObject()) > key ) {  
2      start.setPrev (newElem);  newElem.setNext (start);  
3      newElem.setPrev (null);   start = newElem;  
4      return;  
5  }
```



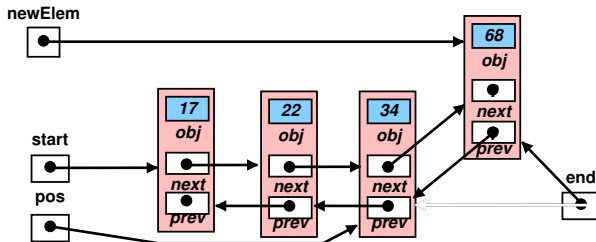
Insert between two elements:

```
1  DLElem pos = start;
2  while ( pos.getNext() != null)
3      if ( getKey(pos.getNext().getObject()) > key ) {
4          pos.getNext().setPrev(newElem);
5          newElem.setNext (pos.getNext());
6          newElem.setPrev (pos);
7          pos.setNext (newElem);
8          return;
9      }
10 else pos = pos.getNext();
```



Insert at end of list:

```
1 pos.setNext(newElem);    end = newElem;  
2 newElem.setNext(null);  newElem.setPrev(pos);
```



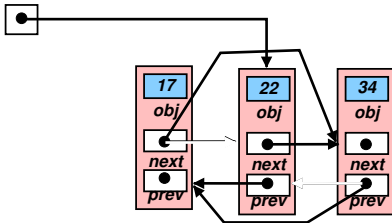
Deletion of arbitrary element,
with special treatment for first/last element:

```
1 public void delete (DLElem elemToDelete) {  
2     if (elemToDelete == null )  
3         return;  
4     if (elemToDelete.getPrev() != null)  
5         elemToDelete.getPrev().setNext(elemToDelete.getNext());  
6     else // special case first element:  
7         start = elemToDelete.getNext();  
8     if (elemToDelete.getNext() != null)  
9         elemToDelete.getNext().setPrev(elemToDelete.getPrev());  
10    else // special case last element:  
11        end = elemToDelete.getPrev();  
12 }
```

Remove from double linkage:

```
1 ...  
2 elemToDelete.getNext().setNext(elemToDelete.getNext());  
3 ...  
4 elemToDelete.getNext().setPrev(elemToDelete.getPrev());  
5 ...
```

elemToDelete



Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- **Inheritance**
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Relationships between objects or classes: Using references, we can refer from one object of a class to another object of the same or a different class.

*References define **relationships** between objects (instance or object level) and between classes (type or class level)*

Types of relationships:

- **organizational**
e.g., a list element refers to its successor
- **application-specific**
e.g., a student is enrolled for a field of study
- **"is-a"** (specialization, generalization, **inheritance**)
e.g., an automobile is a vehicle (and has all properties of a vehicle, in addition to properties specific to an automobile)
- **"has-a"** (composition)
e.g., a car has an engine, has four wheels, etc.

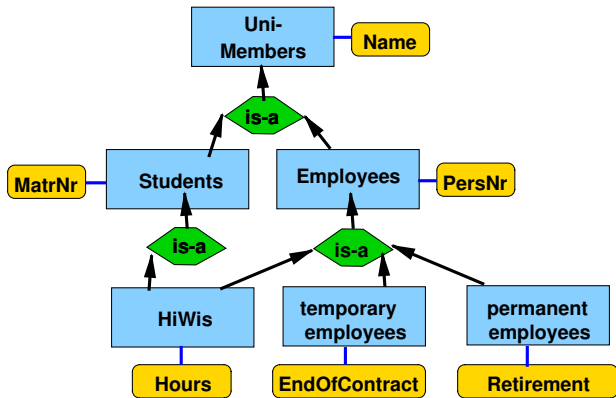
Inheritance:

- Goal: Reusability!
- Definition of new classes based on existing classes
 - ▶ Reuse data and behavior from the existing class
 - ▶ add new properties
- Superclass is **specialized** to subclass
 - ▶ objects with additional, more specific properties.
- Superclass **generalizes** its subclasses
 - ▶ generalization of the objects to their common properties.
- Keyword in Java: subclass “**extends**” superclass
- Principle:
 - ▶ Subclass inherits the variables and methods of the superclass,
 - ▶ extends them by new variables and methods and
 - ▶ may overwrite methods to adapt to the extended properties of the objects.

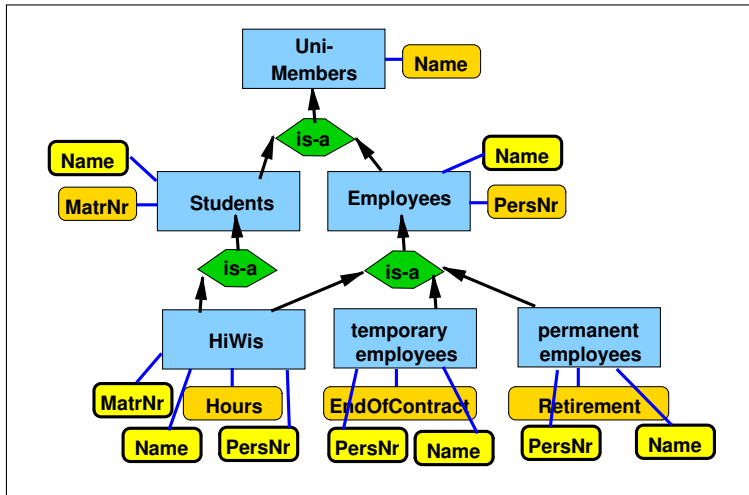
Class hierarchy

- The inheritance defines a **class hierarchy**.
- The most general class in Java is the class **Object**.
- An object of a subclass is also an object of all corresponding superclasses, e.g.:
*A sports car is a car,
is also a land vehicle,
is also a means of transportation,
is also a ...*
 - ▶ **subclass**: sports car
 - ▶ **direct superclass**: car
 - ▶ **indirect superclasses**: land vehicle, means of transportation, ...
- Superclasses usually represent larger sets of objects than subclasses, e.g.:
 - ▶ **superclass** land vehicle includes cars, buses, trucks, motorbikes, bicycles, ...
 - ▶ **subclass** car only contains vehicles with special properties.

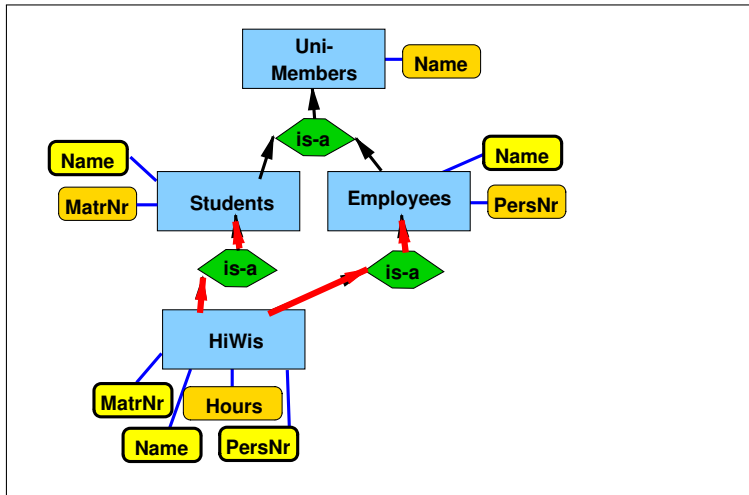
Example: **University members**,
initially without inherited properties:



Example: **University members**,
with inherited properties:



Attention: Example includes **multiple inheritance!**
(forbidden in Java, only 'workaround' possible)

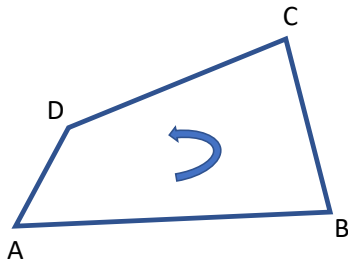


Inheritance hierarchy in Java

- Inheritance hierarchies were developed in the 60ies when trying to process natural language with the computer. Thus inheritance hierarchies often followed terminological is-a relationships.
- Inheritance hierarchies have a tree-like structure, where every class is super- and/or subclass.
- In Java the root of the inheritance hierarchy (i.e., the most general class) is the class **Object**.
- In the following Java examples, reuse of class definitions is in the focus.

Inheritance hierarchy in Java

- We consider specific geometric objects, namely quadrilaterals (polygons with four sides and four corners):
 - ▶ Quadrilaterals
 - ▶ Parallelograms
 - ▶ Rectangles
- We represent a geometric object by its corners, i.e., for a quadrilateral, we store four corner points (in counter-clockwise order)



As a foundation for representing these objects, we use the class `Pt` which represents points in the plane.

```
1 public class Pt {
2     private int x, y;
3
4     public Pt(int xVal, int yVal){
5         x = xVal ; y = yVal ; }
6
7     public void setX( int xVal ) {x = xVal;}
8
9     public void setY( int yVal ) {y = yVal;}
10
11     public int getX() {return x;}
12     public int getY() {return y;}
13
14     public double dist(Pt p){
15         return Math.sqrt((x-p.getX())*(x-p.getX())
16                             +(y-p.getY())*(y-p.getY()));}
17
18     public String toString() {return "("+x+";"+y+"");}
19 }
```

Class for Quadrilateral objects

```
1 public class Quadrilateral {
2     private Pt a,b,c,d;
3
4     public Quadrilateral() { }
5     public Quadrilateral
6         (Pt p1, Pt p2,
7          Pt p3, Pt p4){
8         a=p1; b=p2; c=p3; d=p4;
9     }
10
11
12
13
14
15     public Pt getA() {return a;}
16     public Pt getB() {return b;}
17     public Pt getC() {return c;}
18     public Pt getD() {return d;}
19
20     ...
```

Class for Parallelogram objects

```
1 public class Parallelogram {
2     private Pt a,b,c,d;
3
4     public Parallelogram() { }
5     public Parallelogram
6         (Pt p1, Pt p2,
7          Pt p3){
8         a=p1; b=p2; c=p3;
9         int dx=a.getX()-b.getX();
10        int dy=a.getY()-b.getY();
11        d=new Pt (c.getX()+dx,
12                 c.getY()+dy);
13    }
14
15    public Pt getA() {return a;}
16    public Pt getB() {return b;}
17    public Pt getC() {return c;}
18    public Pt getD() {return d;}
19
20    ...
```

```

1      ...
2
3
4
5
6
7
8
9
10     public double perimeter() {
11         return a.dist(b)+b.dist(c)
12             +c.dist(d)+d.dist(a); }
13
14     public String toString(){
15         return "["+a+","+b+","+
16             +c+","+d+"]"; }
17 }

```

```

1      ...
2     public double area() {
3         return
4             a.getX()*b().getY()
5             +a.getY()*c().getX()
6             +b.getX()*c().getY()
7             -b.getY()*c().getX()
8             -a.getY()*b().getX()
9             -a.getX()*c().getY(); }
10
11     public double perimeter() {
12         return a.dist(b)+b.dist(c)
13             +c.dist(d)+d.dist(a); }
14
15     public String toString(){
16         return "PG["+a+","+b+","+
17             +c+","+d+"]"; }
18 }

```

These *independent* definitions of the classes **Quadrilateral** and **Parallelogram** show commonalities, e.g.:

- equal: variables `a`, `b`, `c`, `d`, methods `getA` etc., `perimeter`
- different are: constructor, methods `area`, `toString`

Now: Parallelogram as subclass of Quadrilateral:

```
1 public class Quadrilateral {
2
3     private Pt a,b,c,d;
4
5     public Quadrilateral() { }
6     public Quadrilateral
7         (Pt p1, Pt p2,
8          Pt p3, Pt p4){
9         a=p1; b=p2; c=p3; d=p4;
10    }
11
12
13
14
15
16
17
18    public int getA() {return a;}
19    public int getB() {return b;}
20    public int getC() {return c;}
21    public int getD() {return d;}
22    ...
```

```
1 public class Parallelogram
2     extends Quadrilateral{
3     // a,b,c,d are inherited
4
5     public Parallelogram() { }
6     public Parallelogram
7         (Pt p1, Pt p2,
8          Pt p3){
9         super(p1,p2,p3,
10             new Pt (p3.getX()
11                     +p1.getX()
12                     -p2.getX(),
13                     p3.getY()
14                     +p1.getY()
15                     -p2.getY()));
16     }
17
18     // getA() is inherited
19     // getB() is inherited
20     // getC() is inherited
21     // getD() is inherited
22     ...
```

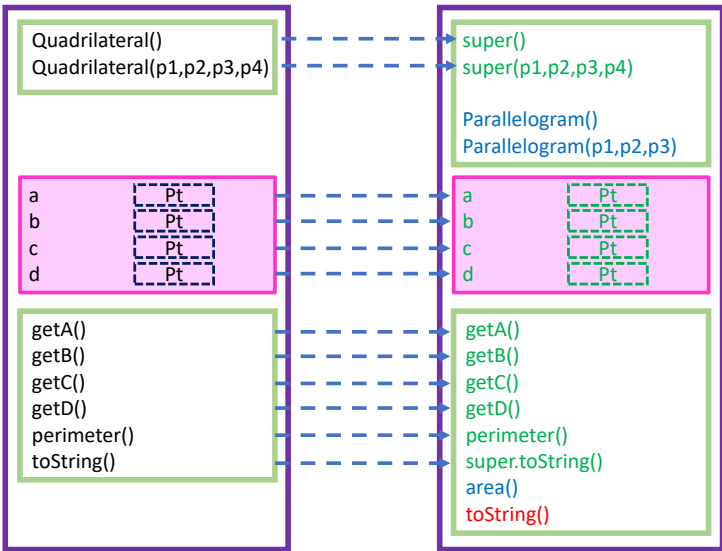
```
1    ...
2
3
4
5
6
7
8
9
10
11 public double perimeter() {
12     return a.dist(b)+b.dist(c)
13         +c.dist(d)+d.dist(a); }
14
15
16 public String toString(){
17     return "["+a+","+b+","+
18         +c+","+d+"]"; }
19 }
```

```
1    ...
2 public double area() {
3     return
4         getA().getX()*getB().getY()
5         +getA().getY()*getC().getX()
6         +getB().getX()*getC().getY()
7         -getB().getY()*getC().getX()
8         -getA().getY()*getB().getX()
9         -getA().getX()*getC().getY(); }
10
11 // perimeter inherited
12
13
14
15 // toString, overwritten
16 public String toString(){
17     return "PG"
18         +super.toString(); }
19 }
```


- **a,b,c** and **d** are declared as **private** in **Quadrilateral** and thus cannot be directly accessed in **Parallelogram**.
- Alternative: Modifier **protected**, between **public** and **private**: accessible in all subclasses and all classes of the same package (see later)
- Three methods of the superclass **Quadrilateral** are not directly accessible in the subclass **Parallelogram**, but only with the keyword **super** to access the superclass:
 - ▶ the two constructors **Quadrilateral()** and **Quadrilateral(p1,p2,p3,p4)**, only via **super()** or **super(p1,p2,p3,p4)** at the start of the constructors of **Parallelogram**.
 - ▶ the overwritten method **toString()** as **super.toString()**

The calls **super.toString()**, **super()** and **super(p1,p2,p3,p4)** can be used in this form only in direct subclasses

- **Quadrilateral** and **Parallelogram** are also (indirect) subclasses of **Object** and thus have additional methods (later...)!



inherited new overwritten

Visibility modifiers for the instance variables **a, b, c, d** in **Quadrilateral**:

- **private**:
then access to **a, b, c, d** from the subclass **Parallelogram** not directly possible, only using inherited methods, i.e.,
getA() , getB() , getC() , getD() , perimeter() ,
super.toString() , super() , super(p1,p2,p3,p4)
- **protected**: direct access from subclass **Parallelogram** allowed
- **public**: direct access allowed in all classes

Recommendation: Variables should be **private**, without direct access outside the defining class

Thus: define **a, b, c** and **d** in **Quadrilateral** as **private** and access them only via methods!

Complete example C5E09_Inheritance:

- file `Pt.java`: **class** `Pt` as before for representing points in the plane.
- file `Quadrilateral.java`: **class** `Quadrilateral` as above, with **private** modifier for the instance variables.
- file `Parallelogram.java`: **Klasse** `Parallelogram` as subclass of `Quadrilateral`.
- file `Rectangle.java`: **extension** of the class hierarchy by the **class** `Rectangle` as subclass of `Parallelogram`.
- additionally **main** in the class `Rectangle` as a method for testing.

file `Rectangle.java`: extension of the class hierarchy by the class `Rectangle` as subclass of `Parallelogram`

```
1 public class Rectangle extends Parallelogram {
2
3     public Rectangle() {}
4
5     public Rectangle( Pt p1, Pt p2, Pt p3) {
6         super( p1, p2, p3 );
7     }
8
9     public double area() {
10         return getA().dist(getB())
11             *getB().dist(getC());
12     }
13
14     public double diag() {
15         return getA().dist(getC());
16     }
17
18     @Override public String toString() {
19         return "Rect:" + super.toString();
20     }
21     ...
```

in file `Rectangle.java`: method for testing class `Rectangle`

```
1  public static void main( String[] args ) {  
2      System.out.println("Test method for class hierarchy.\n");  
3  
4      Rectangle c = new Rectangle  
5                      (new Pt(0,0),new Pt(2,0),new Pt(2,2));  
6  
7      System.out.println(rect);  
8      System.out.println("area   : " +rect.area());  
9      System.out.println("perim  : "+rect.perimeter());  
10     System.out.println("diag   : "+rect.diag());  
11  
12 }  
13 }
```

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- **Chains of constructors**
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

The generation of a subclass object generates a sequence of **constructor** calls:

- The subclass constructor first(!) implicitly or explicitly calls the superclass constructor.
- The constructor calls follow the class hierarchy up to the constructor of the class **Object**.
- The method bodies of the constructors are then executed from the superclass to the subclass (along the class hierarchy 'top down').
- Explicit calls of a constructor of the superclass must always be done at the beginning of the declaration of the calling constructor.

- The **garbage collector** removes objects from memory to which no references refer.
- In early versions of Java, a **finalizer** was recommended, but this has been deprecated since Java 9.
- Many programs never make use of finalizers!
- In practice the garbage collector works only after some delay. i.e., even calling it directly may not immediately start garbage collection.

Example K5B10E_Constructors:

(a) class **SuperClass** for constructor example:

```
1 public class SuperClass {
2
3     private String superData;
4
5     public SuperClass() {
6         superData = "-SP-";
7         System.out.println("super default constructor: " + this);
8     }
9
10    public SuperClass(String name) {
11        superData=name ;
12        System.out.println("super special constructor: " + this);
13    }
14
15    @Override public String toString() {
16        return "superData_" + superData;
17    }
18 }
```

(b) class **SubClass** for constructor example:

```
1 public class SubClass extends SuperClass {
2
3     private String subData;
4
5     public SubClass() {
6         subData = "-sb-";
7         System.out.println("sub default constructor:    " + this);
8     }
9
10    public SubClass(String supname, String subname) {
11        super( supname );
12        subData = subname;
13        System.out.println("sub special constructor:    " + this);
14    }
15
16    @Override public String toString() {
17        return "subData_" + subData + ",_" + super.toString();
18    }
19 }
```

(c) test class:

```
1 public class Test {  
2     public static void main( String args[] ) {  
3         SuperClass a = new SuperClass ("-AA-");  
4         SubClass    b = new SubClass    ();  
5         SubClass    c = new SubClass    ("-C1-", "-C2-");  
6         System.out.println();  
7         a = null;    // marked for garbage collection  
8         b = null;    //           - " -  
9         c = null;    //           - " -  
10        System.gc(); // call garbage collector  
11    } }
```

(d) example output:

```
1 super special constructor: superData -AA-  
2 super default constructor: subData null, superData -SP-  
3 sub default constructor:   subData -sb-, superData -SP-  
4 super special constructor: subData null, superData -C1-  
5 sub special constructor:   subData -C2-, superData -C1-
```

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- **Referencing superclasses and subclasses**
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Important: When can we store a reference to a class in a variable of another class?

Possible combinations of variables and references with inheritance (with example **Parallelogram** as subclass of **Quadrilateral**):

$x = r$	reference r to superclass	reference r to subclass
variable x of superclass type	yes	yes (Parallelogram 'is-a' Quadrilateral)
variable x subclass type	no (Quadrilateral 'is-NOT-a' Parallelogram)	yes

Example C5E11_References (using the known classes Quadrilateral and Parallelogram):

```
1 public class References {
2     public static void main( String[] args ) {
3         Pt a=new Pt(0,0), b=new Pt(1,0), c=new Pt(1,1), d=new Pt(0,1);
4
5         // Superclass reference to Superclass object:
6         Quadrilateral ql = new Quadrilateral(a,b,c,d);
7
8         // Subclass reference to Subclass object:
9         Parallelogram pg = new Parallelogram(a,b,c);
10
11        // Superclass reference to Subclass object:
12        Quadrilateral qlRef = pg;
13
14        // Subclass reference to Superclass object:
15        // Parallelogram pgRef = ql; // compile error
16
17        System.out.println( ql.toString() +
18            "\n_(call toString() of the class Quadrilateral)\n\n");
19
20        System.out.println( pg.toString() +
21            "\n_(call toString() of the class Parallelogram)\n\n");
22
23        System.out.println( qlRef.toString() +
24            "\n_(call toString() of the class Parallelogram)\n\n");
25    }}
```

Casting of a superclass reference is possible if the test **instanceof** for the subclass property is successful.

instanceof is a special boolean operator that has a reference and a class name as parameters, see Example K5B12E_ReferenceCast:

```
1 public class ReferenceCast {
2     public static void main( String[] args ) {
3
4         Pt a=new Pt(0,0), b=new Pt(1,0), c=new Pt(1,1), d=new Pt(0,1);
5
6         Quadrilateral qlRef = new Parallelogram(a,b,c,d);
7
8         Parallelogram pgRef = new Parallelogram();
9         if ( qlRef instanceof Parallelogram )
10             pgRef = (Parallelogram) qlRef;
11
12         System.out.println( qlRef.toString()
13             + "\n__call toString() of the class Parallelogram"
14             + "\n__superclass reference to subclass object\n\n");
15
16         System.out.println( pgRef.toString()
17             + "\n__call toString() of the class Parallelogram"
18             + "\n__superclass reference to subclass object "
19             + "cast to Parallelogram\n");
20     } }
```


Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- **Override annotation**
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

- Annotation: language component to include meta data into the source code, used by the compiler
- most important example: `@Override`
- purpose: annotate methods that overwrite superclass methods
- if no corresponding superclass method exists: compile error!
- helps to identify typos, e.g. `@Override public toString()` instead of `@Override public toString()`
- other annotations: `@Deprecated`

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- **Abstract methods & abstract classes**
- Interfaces
- Inner Classes

Methods in Java can be defined as 'abstract' (without body):

```
1 public abstract double perimeter (); // no '{}'
```

A class including an abstract method must be defined as abstract as well:

```
1 public abstract class Figure {  
2     public abstract double perimeter ();  
3     public abstract double area ();  
4 }
```

- Abstract classes cannot be instantiated, they only serve to construct subclasses.
- Classes can also be defined as abstract if they include non-abstract methods. (This prevents an instantiation.)

- A subclass of an abstract class can be instantiated, if it overwrites and implements all abstract methods of the superclass.
- A subclass that does not implement all inherited abstract methods is itself abstract.

	class	abstract class	final class
Generation of objects	yes	no	yes
Creation of subclasses	yes	yes	no

Example: *abstract* class Shape

```
1 public abstract class Shape {
2
3 // concrete constructor:
4 // for (usually implicit) calls by subclass constructors
5 protected Shape() { }
6
7 // abstract methods:
8
9 public abstract double perimeter();
10
11 // concrete methods:
12
13 public String getName() {
14     return this.getClass().toString();
15     // name of the class of an object, as String
16 }
17 }
```

Quadrilateral as concrete subclass of the abstract class Shape

```
1 public class Quadrilateral extends Shape {
2
3     private Pt a,b,c,d;
4
5     public Quadrilateral() { }
6
7     public Quadrilateral(Pt p1, Pt p2, Pt p3, Pt p4) {
8         a=p1; b=p2; c=p3; d=p4; }
9
10    // overwrite abstract methods with concrete methods
11
12    @Override public double perimeter() {
13        return a.dist(b)+b.dist(c)+c.dist(d)+d.dist(a); }
14
15    // additional methods:
16
17    public int getA() {return a;}
18    public int getB() {return b;}
19    public int getC() {return c;}
20    public int getD() {return d;}
21
22    @Override public String toString(){
23        return "["+a+","+b+","+c+","+d+"]"; }
24 }
```

Parallelogram as indirect subclass of the abstract class Shape

```
1 public class Parallelogram extends Quadrilateral {
2 // new constructors:
3 public Parallelogram() { }
4 public Parallelogram (Pt p1, Pt p2, Pt p3) {
5     super(p1,p2,p3, new Pt (p3.getX()+p1.getX()-p2.getX(),
6                             p3.getY()+p1.getY()-p2.getY()));
7 }
8 // new methods:
9 public double area() {
10     return getA().getX()*getB().getY()+getA().getY()*getC().getX()
11            +getB().getX()*getC().getY()-getB().getY()*getC().getX()
12            -getA().getY()*getB().getX()-getA().getX()*getC().getY();
13 }
14 // inherited (from Shape): getName
15
16 // inherited (from Quadrilateral): perimeter
17
18 // overwritten (from Quadrilateral):
19 @Override public String toString() {
20     return "PG" + super.toString();
21 }
22 }
```


Rectangle as indirect subclass of the abstract class Shape

```
1 public class Rectangle extends Parallelogram {
2     // new:
3     public Rectangle() { }
4     public Rectangle(Pt p1, Pt p2, Pt p3 ) {
5         super( p1, p2, p3 );
6     }
7
8     public double diag() {
9         return getA().dist(getC());
10    }
11    // overwritten (from Parallelogram):
12    @Override public double area() {
13        return getA().dist(getB())
14            *getB().dist(getC());
15    }
16    }
17    @Override public String toString() {
18        return "Rect:"+super.toString()
19    }
20    // inherited (from Quadrilateral): perimeter
21    // inherited (from Shape): getName
22 }
```

Example C5E13_Shape_Abstract: calling 'polymorphic' methods

```
1 public class ShapeTest {
2     public static void main( String args[] ) {
3
4         Quadrilateral ql = new Quadrilateral(new Pt(0,0),new Pt(1,0),
5                                             new Pt(1,1), new Pt(0,1));
6         Parallelogram pg = new Parallelogram(new Pt(0,0),new Pt(2,0),
7                                             new Pt(2,2));
8         Rectangle re = new Rectangle(new Pt(0,0),new Pt(3,0),
9                                     new Pt(3,3));
10
11 // usual access to methods of each class:
12 System.out.println("\n\n" + ql.getName() + "_(direct)\n"
13                     + "[_" + ql.getA() + "_,_" + ql.getB()
14                     + "_,_" + ql.getC() + "_,_" + ql.getD() + "]" );
15
16 System.out.println("\n\n" + pg.getName() + "_(direct)\n"
17                     + "[_" + pg.getA() + "_,_" + pg.getB()
18                     + "_,_" + pg.getC() + "_,_" + pg.getD() + "]" );
19
20 System.out.println("\n\n" + re.getName() + "_(direct)\n"
21                     + "[_" + re.getA() + "_,_" + re.getB()
22                     + "_,_" + re.getC() + "_,_" + re.getD() + "]" );
23
24 ...
```

```
1  ...
2  Shape arrayOfShapes[] = new Shape[ 3 ];
3  arrayOfShapes[ 0 ] = ql;assignment of subclass references!
4  arrayOfShapes[ 1 ] = pg;
5  arrayOfShapes[ 2 ] = re;
6  // access via concrete methods of Shape/Quadrilateral:
7  for ( int i = 0; i < arrayOfShapes.length; i++ ) {
8      System.out.println("\n\n"
9          + arrayOfShapes[ i ].getName() + "_ (via_Shape)\n");
10 }
11 // use of overwritten methods:
12 for ( int i = 0; i < arrayOfShapes.length; i++ ) {
13     System.out.println("\n\n"
14         + arrayOfShapes[ i ].getName() + "_ (via_Shape)\n"
15         + "\n" + arrayOfShapes[ i ].toString()
16         + "\nperimeter = " + arrayOfShapes[ i ].perimeter());
17 }
18 }
19 }
```

Polymorphism:

- Methods are **polymorphic** if they can be used with objects of different types.
- Implementation in Java:
 - 1 Definition the method in a superclass (abstract or not)
 - 2 Overwriting of the method in the corresponding subclasses.
 - 3 Method can be applied for objects of classes where the implementation of the method is defined.
- The array **arrayOfShapes** is defined on the abstract class **Shape** and can thus contain elements of arbitrary (non-abstract) subclasses of **Shape**.
- The methods **perimeter**, **getName** can be called for the abstract class **Shape**, the (overwriting) implementation of the corresponding subclass is executed.
- The methods **getA**, **getB**, **getC**, **getD** can be called only for objects of the class **Quadrilateral** or its subclasses. The method **area** and **toString** are polymorphic since the implementations in different subclasses are different.

Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- **Interfaces**
- Inner Classes

Interfaces are similar to abstract classes and are used to define interfaces.

	Interfaces	abstract classes
identifier	interface ifname	abstract class classname
methods	all implicitly abstract	abstract and concrete
variables	only static final	arbitrary
constructors	none	default or explicit
subclasses	implements ifname	extends classname
subinterfaces	extends ifname	-

A class can

- be a subclass of *at most one* superclass (extends), but
- implement *arbitrary many* interfaces (implements)

Thus:

Interfaces in Java implement a restricted form of multiple inheritance.

Example interface **Shape**

- no constructor (constructor not useful since any variables must be `static final`)
- All methods are implicitly(!) defined as **abstract**.

```
1 public interface Shape {  
2  
3     // no constructor !  
4     // all methods abstract  
5  
6     public double perimeter();  
7     public String getName();  
8 }
```

Quadrilateral as concrete subclass of interface Shape:

- overwrites abstract methods with concrete (implemented) methods

```
1 public class Quadrilateral implements Shape {
2
3     private Pt a,b,c,d;
4
5     public Quadrilateral() { }
6     public Quadrilateral(Pt p1, Pt p2, Pt p3, Pt p4) {
7         a=p1; b=p2; c=p3; d=p4; }
8
9     //implementation as before:
10    public int getA() {return a;}
11    public int getB() {return b;}
12    public int getC() {return c;}
13    public int getD() {return d;}
14
15    //implementation of abstract methods:
16    @Override public double perimeter() {
17        return a.dist(b)+b.dist(c)+c.dist(d)+d.dist(a); }
18    @Override public String getName() { return this.getClass().toString() }
19    //overwrite the method inherited from Object :
20    @Override public String toString()
21        {return "["+a+","+b+","+c+","+d+"]"; }
22 }
```


The classes `Parallelogram`, `Rectangle` and `ShapeTest` can be reused without any change(!) from the example for abstract classes with identical result, see `C5E14_Shape_Interface!`

6 Exceptions

- Introduction
- Handling Exceptions
- Checked & Unchecked Exceptions

- For some problems at runtime, the Java Virtual Machine generates (*throws*) special objects describing the observed exceptional situation.
- These objects belong to subclasses of **Error** and **Exception**.
- **Error** and **Exception** are part of the package `java.lang` and are subclasses of **Throwable**.
- **Exception** objects can be *caught* and handled in the program.
- **Error** objects represent situations that the program cannot repair itself.

- **Error** has the subclasses
 - ▶ **LinkageError**
 - ▶ **ThreadDeath**
 - ▶ **VirtualMachineError**
 - ▶ **AWTError**
 - ▶ ...
- **Exception** has the subclasses:
 - ▶ **RuntimeException** mit den Unterklassen
 - ★ **ArithmeticException**
 - ★ **IndexOutOfBoundsException**
 - ★ **NullPointerException**
 - ★ **NoSuchElementException**
 - ★ ...
 - ▶ **NoSuchMethodException**
 - ▶ **ClassNotFoundException**
 - ▶ **IOException**
 - ▶ ...

6 Exceptions

- Introduction
- **Handling Exceptions**
- Checked & Unchecked Exceptions

Exceptions can be handled in three ways:

- The program does not handle the exception.
- Exceptions are caught where they are generated.
- Exceptions are caught somewhere else in the program.

If an exception is not caught, the program terminates with an output on the standard error output (**`System.err`**) with details about:

- the type of exception
- the location in the program where the exception was thrown

No handling of exceptions:

```
1 public class K6B01E_UncaughtException {  
2  
3     public static void main (String[] args) {  
4         int result = 1 / 0;  
5         System.out.println("This output is not shown" );  
6     }  
7  
8 }
```

Division by 0 generates **ArithmeticException**:

```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero  
2     at K6B01E_UncaughtException.main(K6B01E_UncaughtException.java:4)
```

Catching exceptions where they are generated:

```
1 public class K6B02E_CaughtException {
2     public static void main (String[] args) {
3         try {
4             String str = System.console().readLine("denominator = ");
5             int denominator = Integer.parseInt(str);
6             int int result = 1 / denominator;
7             System.out.println("without error: output 1");
8         }
9         catch (ArithmeticException newAE) {
10             System.out.println("Arithmetic_Exception!");
11         }
12         ...
13     }
14 }
```

input	reaction
1	without error: output 1
0	Arithmetic Exception!
to	program termination


```
1 ...  
2 try {  
3     String str = System.console().readLine("denominator = ");  
4     int denominator = Integer.parseInt(str);  
5     int result = 1 / denominator;  
6     System.out.println("without error: output 2");  
7 }  
8 catch (ArithmeticException newAE) {  
9     System.out.println(newAE.toString());  
10 }  
11 catch (NumberFormatException newNFE) {  
12     System.out.println(newNFE.toString());  
13 }  
14 finally {  
15     System.out.println("always: output 3");  
16 }  
17 }  
18 }
```

- **try** block without error: continue with (optional) **finally** block.
- exception in **try** block:
 - ▶ if matching **catch** block exists:
continue with this **catch** block, then continue with **finally** block.
 - ▶ if no matching **catch** block exists:
continue with **finally** block (**then program termination!**)

Handling exceptions somewhere else:

- If an exception is not caught within a method, then:
 - ▶ the method execution is aborted immediately;
 - ▶ the exception is propagated to the calling method
- The calling method can catch the propagated exception only if the propagating method was executed within a **try** block.
- The propagation continues until the exception is either caught or propagated outside the method **main**.
The latter causes a program termination.

Exceptions over multiple call levels and classes:

```
1 public class K6B03E_Propagation {
2     public static class Scope { // nested static class...
3
4         public void level1 () {
5             System.out.println("Level_1: Start");
6             int result = 1 / 0;
7             System.out.println("Level_1: End");
8         }
9
10        public void level2 () {
11            System.out.println("Level_2: Start");
12            level1 ();
13            System.out.println("Level_2: End");
14        }
15
16        ...
17    }
```

```
1  ...
2  public void level3() {
3      System.out.println("Level_3:_Start");
4      try { level2(); }
5      catch (ArithmeticException newAE) {
6          System.out.print ("Exception_message:_");
7          System.out.println(newAE.getMessage());
8          System.out.print("Exception_(toString):_");
9          System.out.println(newAE.toString());
10         System.out.println("Call_stack_trace:-----");
11         StackTraceElement[] stackTrace = newAE.getStackTrace();
12         for (int i = 0; i < stackTrace.length; i++)
13             System.out.println("_" + stackTrace[i].toString());
14         System.out.println("-----");
15     }
16     System.out.println("Level_3:_End");
17 }
18 }
```

Test of the propagation over multiple call levels:

```
1  public static void main (String[] args) {  
2      Scope demo = new Scope();  
3      System.out.println("Program: Start");  
4      demo.level3();  
5      System.out.println("Program: End");  
6  } }
```

```
1  Program: Start  
2  Level 3:  Start  
3  Level 2:  Start  
4  Level 1:  Start  
5  Exception message:      / by zero  
6  Exception (toString):  java.lang.ArithmeticException: / by zero  
7  Call stack trace:-----  
8      K6B03E_Propagation$Scope.level1 (K6B03E_Propagation.java:13)  
9      K6B03E_Propagation$Scope.level2 (K6B03E_Propagation.java:19)  
10     K6B03E_Propagation$Scope.level3 (K6B03E_Propagation.java:25)  
11     K6B03E_Propagation.main (K6B03E_Propagation.java:6)  
12  -----  
13  Level 3:  End  
14  Program:  End
```

6 Exceptions

- Introduction
- Handling Exceptions
- Checked & Unchecked Exceptions

- Some exceptions can be expected and thus handled in a natural way, others cannot be ‘repaired’...
- Java thus offers two types of exceptions:
 - ▶ **Checked Exceptions** for exceptions that can be expected and can be handled by the user (and must be handled)
 - ▶ **Unchecked Exceptions** for the hopeless cases
- **The JAVA Tutorials**(Oracle):
 - If a client can reasonably be expected to recover from an exception, make it a checked exception.*
 - If a client cannot do anything to recover from the exception, make it an unchecked exception.*
- but for example Bruce Eckel (in “**Thinking in Java**”) states:
Checked Exceptions are not needed at all...

- **Checked Exceptions** must be either caught within a method or declared in the **throws** clause of the method header.
- **Unchecked Exceptions** do not need to be declared in the **throws** clause; only objects of the class **RuntimeException** are unchecked, e.g.:

ArithmeticException
BufferOverflowException
BufferUnderflowException
ClassCastException
EmptyStackException
IllegalArgumentException
IndexOutOfBoundsException
MissingResourceException
NegativeArraySizeException
NoSuchElementException
NullPointerException
NumberFormatException
...

Example: `java.io.File` includes methods for accessing files, that can be used with the `Scanner` class similar to `System.in`.

Difference: `System.in` always exists, but files could be missing or unreadable. Thus, when instantiating `Scanner` objects, very critical errors may happen that *must* be handled, see constructor for `Scanner` objects from `File`:

```
1 public Scanner(File source) throws FileNotFoundException
```

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4 public class K6B05E_File{
5     public static void main(String[] args)
6         throws FileNotFoundException {
7         Scanner sc = new Scanner(System.in);
8         System.out.print ("Input file name: ");
9         String filename = sc.nextLine();
10        File file = new File(filename);
11        Scanner input = new Scanner(file);
12        while ( input.hasNextLine() ){
13            System.out.println(input.nextLine());
14        }    }
```

Alternative solution:

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 public class K6B06E_FileCheck{
6     public static void main(String[] args)  {
7
8         Scanner sc = new Scanner(System.in);
9         Scanner input;
10
11         while ( true ) {
12             System.out.print ("Input file name: ");
13             String filename = sc.nextLine();
14             File file = new File(filename);
15             try{
16                 input = new Scanner(file);
17                 break;
18             } catch ( FileNotFoundException fnfe ) {
19                 System.out.println ("\nError: file not found!");
20             }
21         }
22
23         while ( input.hasNextLine() ){
24             System.out.println(input.nextLine());
25         }
26     } }
```

7 Generic Programming

- Generics
- Collection Framework

Generics: methods and classes can have *types as parameters*.

- Another form of defining polymorphic methods
- Alternative to **Interface** for implementing comparable accesses with different classes.
- Goal: generic formulation of algorithms that will be later instantiated depending on the type.
- Syntax with methods: type parameters are given before the return type of the method

```
1 public <T> T doSomething ( T[] array, int i ) {  
2     return array [i];  
3 }  
4 ...  
5 Integer [] IntegerArray;  
6 ...  
7 doSomething (IntegerArray, 5);  
8 ...  
9 Double [] DoubleArray;  
10 ...  
11 doSomething (DoubleArray, 4);
```

Example with generic method:

```
1 public class K7B01E_GenericMethod {
2
3     public static < T > String convert( T[] array ) {
4         String result="|";
5         for ( int i=0; i< array.length; i++)
6             result += "_" + array[i] + "_|";
7         return result;
8     }
9
10    public static void main( String args[] ) {
11        Integer[]    iArray = {3, 5, 7 };
12        Double[]     dArray = { 1.1, 2.2, 3.3, 4.4 };
13        Character[]  cArray = { 'H', 'E', 'L', 'L', 'O' };
14
15        System.out.println(
16            convert( iArray ) + "\n\n" +
17            convert( dArray ) + "\n\n" +
18            convert( cArray ) );
19    }
20 }
```

Why Generics?

```
Object o = "String";  
String s = (String) o;
```

- Explicit type cast: object `o` must include `String` object.
- This is ok here...

But:

```
Object o = Integer.valueOf( 42 );    // or Autoboxing: o = 42;  
String s = (String) o;
```

- Type cast implies check for class compatibility at run time, thus here: run time error...
- Alternatives: catch exception or use **`instanceof`**
- Better: check at compile time, then no need for test at run time, run time errors impossible.

~> **Generics**

Example: generic method for maximum,
java.lang includes **interface Comparable<T>**

```
1 public class K7B02E_GenericMaximum {
2
3     public static <T extends Comparable<T> >
4         T maximum( T x, T y, T z ){
5         T max = x;
6         if ( y.compareTo( max ) > 0 )
7             max = y;
8         if ( z.compareTo( max ) > 0 )
9             max = z;
10        return max;
11    }
12
13    public static void main( String args[] ) {
14        System.out.println(
15            maximum(8, 6, 4) + " "
16            + maximum(1.1, 7.7, 4.4) + " "
17            + maximum( "Pear", "Apple", "Orange" ) );
18    }
19 }
```

- At compilation time, the type variables are removed ('erasure') and replaced by concrete types.
- what remains are 'objects' (i.e., **Object** as base type); for a generic type **<T1 extends T2>** the base type **T2** is chosen..
- The purpose of generics is especially checking of type safety during compilation time.

```
1 public static <T extends Comparable<T> >
2     T maximum( T x, T y, T z ){
3     T max = x;
4     if ( y.compareTo( max ) > 0 ) max = y;
5     if ( z.compareTo( max ) > 0 ) max = z;
6     return max;
7 }
```

is compiled to:

```
1 public static
2     Comparable maximum (Comparable x, Comparable y, Comparable z ){
3     Comparable max = x;
4     if ( y.compareTo( max ) > 0 ) max = y;
5     if ( z.compareTo( max ) > 0 ) max = z;
6     return max;
7 }
```


Syntax of the header of **Generic Classes**:

```
class name<T1, T2, ..., Tn> ... {...}
```

Style convention recommends parameter names for generic parameters:

- **E** - Element (esp. with Java Collections Framework, see later)
- **K** - Key
- **N** - Number
- **T** - Type (general type)
- **V** - Value
- **S, U, V** etc. – second, third, fourth type...

Example: generic stack (as static class in K7B03E_GenericStack)

```
1  public static class GenericStack< E > {
2      private int top;
3      private E[] elements;
4
5      public GenericStack() {
6          top=-1;
7          elements = ( E[] ) new Object[ 10 ];
8          //elements = new E[ 10 ];
9      }
10
11     public boolean isFull(){return (top == 9);}
12     public boolean isEmpty(){return (top == -1);}
13     public void push(E value){elements[++top] = value;}
14     public E pop(){return elements[top--];}
15 }
```

compiler warning at:

```
elements = ( E[] ) new Object[ 10 ];
```

System cannot guarantee type safety, here only as warning!

```
Note: K7B03E_GenericStack.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

(obvious) alternative does not work:

```
elements = new E[ 10 ];
```

with error message

```
K7B03E_GenericStack.java:10: error: generic array creation  
    elements = new E[ 10 ];
```

The warning can be ignored here since the array is accessed only through **push** and **pop**, but no external method modifies the array and could insert objects of wrong types.

```
1 public class K7B03E_GenericStack {
2     public static class GenericStack< E > {...} \\s.o.
3
4     public static void main( String args[] ) {
5         double[] dElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
6         int[] iElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
7
8         GenericStack < Double > dStack
9             = new GenericStack< Double >();
10
11         GenericStack < Integer > iStack
12             = new GenericStack< Integer >();
13
14         for ( int i=0; i< dElements.length; i++)
15             if (!dStack.isFull()) dStack.push( dElements[i] );
16         while (!dStack.isEmpty()) System.out.println( dStack.pop() );
17
18         for ( int i=0; i< iElements.length; i++)
19             if (!iStack.isFull()) iStack.push( iElements[i] );
20         while (!iStack.isEmpty()) System.out.println( iStack.pop() );
21     }
22 }
```

Since version 7, Java provides *type inference*: Java tries to determine the types from the context.

Then the generic type '`<>`' ('Diamond') is sufficient, e.g.:

```
1 ArrayList<Double> al = new ArrayList<>();  
2  
3 al.add(new Double(1.1));  
4 al.add(new Double(2.2));  
5 al.add(new Double(3.3));  
6 for (int i = 0; i < al.size(); i++) {  
7     System.out.println(a[i]);  
8 }
```

7 Generic Programming

- Generics
- Collection Framework

A 'collection' is an aggregation of similar data elements with operations for navigating and/or direct access to the individual elements. The Java language itself only provides arrays for this purpose.

The Collection Framework, part of the package `java.util`, supports additional collections in the form of a collection of

- interfaces for typical data structures (e.g. `Queue`, `List`, `Set`), for sequential access (`Enumeration`, `Iterator`, `ListIterator`) and the comparison of elements (`Comparator`).
- implementations of the interfaces with different underlying data structures (e.g. `ArrayList`, `Vector`, `Stack` on dynamically extended arrays, `LinkedList` on a doubly linked list).
- algorithms for typical 'higher' operations, e.g. `sort`, `binarySearch`, `shuffle`, `max`, `min` in the class `Collections` (for `ArrayList`, `Vector`, `Stack`, `LinkedList`, ...) or `sort`, `binarySearch` in the class `Arrays` (for `Arrays`).

- **ArrayList** and **LinkedList** are dynamic sequential data structures.
- Both provide operations expected for **Array**, **Queue**, **Stack** and **List**.
- Both can grow and shrink on demand.
- **ArrayList** is implemented with dynamic arrays. The operations are very efficient (exception: insertion and deletion inner elements).
- **LinkedList** is implemented with a doubly linked list. The operations are usually less efficient than with **ArrayLists**, but inner insertions and deletions are more efficient.
- **Vector** corresponds to **ArrayList**, but is synchronized (suitable for accessing from concurrent threads).
- **Stack** is a **Vector** with explicit **push** and **pop** operations.
- There are many more classes: **PriorityQueue**, **HashSet**, **TreeSet**, **EnumMap**, **HashMap**, ...

Since Java 5.0, the Collection Framework supports generics.

- Lists

- ▶ **ArrayList** List functionality by mapping to an array
- ▶ **LinkedList** doubly linked list

- Sets

- ▶ **HashSet** implements the interface **Set** with a fast hashing method.
- ▶ **TreeSet** implements **Set** with a tree that keeps the elements sorted.
- ▶ **LinkedHashSet** A fast set implementation that in addition also stores the insertion order of the elements.

- Associative Memory

- ▶ **HashMap** implements an associative memory with a hash method.
- ▶ **TreeMap** Instances of this class keep their elements sorted in a binary tree; implements **SortedMap**.
- ▶ **LinkedHashMap** A fast associative memory that additionally stores the insertion order of its elements.

- Queue

- ▶ **LinkedList** The linked list also implements **Queue**.
- ▶ **ArrayBlockingQueue** A blocking queue.
- ▶ **PriorityQueue** A priority queue.

Short overview of important **Collection** methods:

```
interface java.util.Collection<E> extends Iterable
```

- **Enumeration**: only **hasMoreElement()** and **nextElement()**, i.e., enumerated data structure is not changed.
- **Iterator**: **hasNext()**, **next()** and additionally **remove()**

For a Collection **c** with base type **E** the following loops are possible (among others):

```
Iterator<E> it = c.iterator();  
while ( it.hasNext() ){ E e = it.next(); ...}
```

(here, **c** is not changed) or (for-each-loop, see later):

```
for( E e : c ){ something with e...}
```

Iteration with deletion:

```
Iterator<E> it = c.iterator();  
while ( it.hasNext() ){ E e = it.next(); it.remove(); ...}
```

(in this example, **c** is empty at the end)

```
boolean add( E obj )
```

Optional. Adds an element to the container and returns **true** if it was successfully inserted. Returns **false** if an object with the same value is already included and duplicate values are forbidden.

```
void clear()
```

Optional. Deletes all elements in the container.

```
boolean contains( Object obj )
```

Returns **true** if the container includes an element equal to **obj**.

```
boolean isEmpty()
```

Returns **true** if the container does not include any elements.

```
int size()
```

Returns the number of elements in the container.

```
Iterator<E> iterator()
```

Returns **Iterator** object for iterating over all elements of the container.

```
boolean remove( Object obj )
```

Optional. Removes **obj** from the container if it is included.

```
Object[] toArray()
```

Returns array with all elements of the container.

```
<T> T[] toArray( T[] arr )
```

Returns typed array with all elements of the container. Uses **arr** if it is large enough; otherwise, a new array of sufficient size is created.

```
boolean equals( Object obj )
```

Checks if **obj** is also a container and includes the same elements

```
boolean addAll( Collection<? extends E> coll )
```

Adds all elements from the collection **coll** to the container.

```
boolean containsAll( Collection<?> coll )
```

Returns **true** if the container contains all elements of the collection **coll**.

```
boolean removeAll( Collection<?> coll )
```

Optional. Removes all objects from the collection **coll** from the container.

```
boolean retainAll( Collection<?> coll )
```

Optional. Removes all objects that are not contained in collection **coll**.

```
int hashCode()
```

Returns the hash value of the container.

Example: ArrayList <T> for String

```
1 import java.util.*;
2 public class K7B04E_ArrayListString {
3     public static void main(String[] args) {
4
5         ArrayList <String> array = new ArrayList <> ();
6
7         String numbers = System.console().readLine();
8         StringTokenizer tokens = new StringTokenizer (numbers);
9
10        while (tokens.hasMoreTokens () )
11            array.add ( tokens.nextToken () );
12
13        System.out.println("unsorted: " + array);
14
15        Collections.sort(array);
16
17        System.out.println("sorted:   " + array);
18    }
19 }
20 }
```

The compiler checks that **array.add(...)** inserts only objects of type **String**!

Example: ArrayList <T> for Integer

```
1 import java.util.*;
2 public class K7B05E_ArrayListInteger {
3     public static void main(String[] args) {
4
5         ArrayList <Integer> array = new ArrayList <> ();
6
7         String numbers = System.console().readLine();
8         StringTokenizer tokens = new StringTokenizer (numbers);
9
10        while (tokens.hasMoreTokens () )
11            array.add ( Integer.valueOf(tokens.nextToken () ) );
12
13        System.out.println("unsorted: " + array);
14
15        Collections.sort(array);
16
17        System.out.println("sorted:   " + array);
18    }
19 }
20 }
```

The compiler checks that `array.add(...)` inserts only objects of type `Integer`.

Miscellaneous

- 'for each' Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

'for each' loops simplify processing of individual elements of arrays and collections:

Simple example with arrays:

```
1 public class K8B01E_ForEachArray {
2
3     public static void main (String[] args) {
4
5         int day = 0;
6
7         System.out.println("Days of the week:");
8
9         String [] week = { "Monday", "Tuesday", "Wednesday",
10                             "Thursday", "Friday", "Saturday", "Sunday"};
11
12         for (String str : week) //for each String str in week do ...
13             System.out.println( ++day +"th day of the week  = " + str);
14
15     }
16 }
```

‘for each’ loops with multidimensional arrays:

```
1 class K8B02E_ForEachMatrix {
2     public static void main(String[] args) {
3         int [] [] array = { {22, 45, 57, 33},
4                             {64, 28, 19},
5                             {},
6                             {97},
7                             {88, 73, 44, 35, 84} };
8         System.out.println("array (row-wise:");
9         for ( int [] row: array ) {
10             for ( int element : row )
11                 System.out.print( element + "_");
12             System.out.println();
13         }
14     }
15 }
```

‘for each’ loop with collections:

```
1 import java.util.*;
2
3 public class K8B03E_ForEachCollection {
4     public static <E> void printCollection
5         (Collection <E> c, String s) {
6         System.out.println(s + ":");
7         for (E e : c) System.out.println(e);
8         System.out.println("-----");
9     }
10
11     public static void main(String[] args) {
12         LinkedList <Integer> list = new LinkedList<>();
13         list.add(1);
14         list.add(22);
15         list.add(333);
16
17         printCollection(list, "data");
18     }
19 }
```

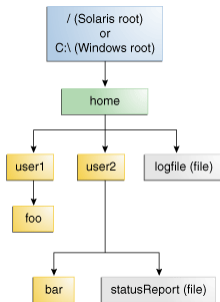
‘for each’ can only be applied for reading:

```
1 public class K8B04E_ForEachRW {
2
3     public static void main (String args[]) {
4
5         String output = "";
6         int [] iArray = new int [5];
7         int count = 0;
8
9         for (int i = 0; i < 5 ; i ++ )
10             iArray [i] = 2*i;  // modification
11
12         for (int i : iArray)
13             System.out.println(i); // read-only access
14
15     }
16 }
```

Miscellaneous

- 'for each' Loops
- **File I/O**
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Tree structure of the file system



source: <https://docs.oracle.com/javase/tutorial/essential/io/path.html>

A **file system** consists of at least one **root**, a hierarchical tree structure of **directories** and **files** (as leaf nodes without further children).

Files and directories are identified using their **path name**, e.g.

/home/sally/statusReport (Solaris, Linux) or

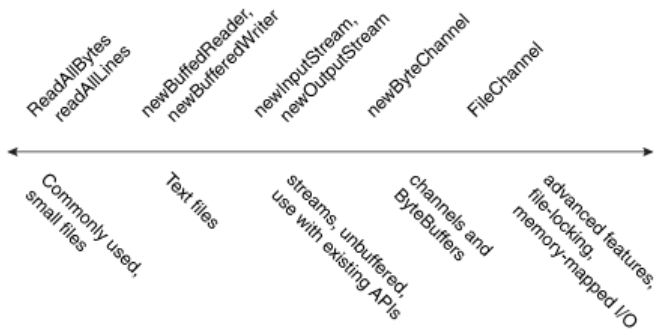
C:\home\sally\statusReport (Windows) – separator character is system-specific in Java!

For the file system, files are sequences of bytes without any further semantics.

In Java, files are abstracted as a **stream** of bytes that are read or written. Additional methods allow to write more complex objects, especially Strings.

“Prior to Java SE 7, File IO involved a fair amount of code, some of it nonintuitive and difficult to remember even for Java veterans. For instance, I would often find myself searching my older code and cutting/pasting into new projects.”

(Eric Bruno, www.drdobbs.com/jvm/java-se-7-new-file-io/231600403)



<https://docs.oracle.com/javase/tutorial/essential/io/file.html>

In the following: recipe for **Java SE 7 – New File IO**

Reading files

- Open an existing file using a path

```
Path path = FileSystems.getDefault().getPath(".", name);
```

- Small files can be read completely:

```
byte[] filearray = Files.readAllBytes(path);
```

- Alternative: read them row-wise, e.g. into **List<String>**:

```
List<String> lines = Files.readAllLines(path,  
    Charset.defaultCharset() );
```

- For big files it is better to read them in pieces:

```
BufferedReader nbr =  
    Files.newBufferedReader(path, Charset.defaultCharset() );  
...  
String line = null;  
while ( (line = nbr.readLine()) != null ) { /* ... */ }  
nbr.close(); // but: see below!
```

the file should be closed, see below!

Writing files

- Creating and writing a new file based on existing data:

```
String content = ...
Files.write( path, content.getBytes(),
            StandardOpenOption.CREATE);
// creates new file.
```

- If the file should be created from small pieces, one can use the following approach:

```
BufferedWriter nbw =
    Files.newBufferedWriter( path, Charset.defaultCharset(),
        StandardOpenOption.CREATE);
...
while ( ... ) {
    String content = ...
    nbw.write(content, 0, content.length());
}
...
nbw.close(); // but: see below!
```

The file must be closed! Alternative to **close**: see below

There are many more possible values for `StandardOpenOption` like `APPEND`, `TRUNCATE_EXISTING` (see documentation!)

Excursion: **try-with-resources**

Alternative to **try-catch-finally** with slightly handling of exceptions in **try** and in **finally/close**

```
try ( BufferedReader nbr = ... )  
{ ... }  
catch(IOException e)  
{...}
```

Here, **nbr.close()** is called automatically at the end of the **try** block, even if an exception is thrown before.

Example program `K8B05E_FileIO.java`:

- Runnable program that uses the methods above
- including `try-catch` where useful
- `try-with-resources` makes sure that the used file is closed at the end of the block.

Used imports:

```
1 import java.io.BufferedReader;
2 import java.io.BufferedWriter;
3 import java.io.IOException;
4 import java.nio.charset.Charset;
5 import java.nio.file.FileSystems;
6 import java.nio.file.Files;
7 import java.nio.file.Path;
8 import java.nio.file.StandardOpenOption;
9 import java.util.ArrayList;
10 import java.util.List;
```

Methods for reading small files:

```
1  public static byte[] readSmallFileBytes(String name) {
2      try {
3          Path path = FileSystems.getDefault().getPath(".", name);
4          return Files.readAllBytes(path);
5      }
6      catch ( IOException ioe ) { ioe.printStackTrace(); }
7      return null;
8  }
9
10 public static List<String> readSmallFileLines(String name) {
11     try {
12         return Files.readAllLines(
13             FileSystems.getDefault().getPath(".", name),
14             Charset.defaultCharset() );
15     }
16     catch ( IOException ioe ) { ioe.printStackTrace(); }
17     return null;
18 }
```

Reading a large file (here into a **List<String>**, close via **try-with-resources**):

```
1  public static List<String> readLargeFileLines(String name) {
2      try ( BufferedReader nbr =
3          Files.newBufferedReader(
4              FileSystems.getDefault().getPath(".", name),
5              Charset.defaultCharset() )
6      ){
7          List<String> lines = new ArrayList<>();
8          while (true){
9              String line = nbr.readLine();
10             if ( line == null ) return lines;
11             lines.add(line);
12         }
13
14     }
15     catch ( IOException ioe ) { ioe.printStackTrace(); }
16     return null;
17 }
```

Writing small files in one shot:

```
1 public static void writeFileBytes(String name, String content){  
2     try {  
3         Files.write(  
4             FileSystems.getDefault().getPath(".", name),  
5             content.getBytes(),  
6             StandardOpenOption.CREATE);  
7     }  
8     catch ( IOException ioe ) { ioe.printStackTrace(); }  
9 }
```


Writing large files piecewise (here from a **List<String>**, again closed using **try-with-resources**):

```
1  public static void writeLargeFileLines(String name,
2                                     List<String> lines) {
3      try ( BufferedWriter nbw =
4              Files.newBufferedWriter(
5                  FileSystems.getDefault().getPath(".", name),
6                  Charset.forName("UTF-8"),
7                  StandardOpenOption.CREATE)
8      ) {
9          for ( String line : lines ) {
10             nbw.write(line, 0, line.length());
11             nbw.newLine();
12         }
13     }
14     catch ( IOException ioe ) { ioe.printStackTrace(); }
15 }
```

Corresponding **main** method:

```
1 public static void main(String[] args) {
2     String outString = "line 1
3     nline 2
4     nline 3";
5     List<String> lines = null;
6     // write bytes in small file
7     writeFileBytes("K8B05a.txt", outString);
8     // read bytes from small file
9     System.out.println("\n--_TEST_1_-----");
10    String inString = new String(readSmallFileBytes("K8B05a.txt"));
11    System.out.println(inString);
12    // read lines from small file
13    System.out.println("\n--_TEST_2_-----");
14    lines = readSmallFileLines("K8B05a.txt");
15    for ( String line: lines ) System.out.println(line);
16    // read lines from large file
17    System.out.println("\n--_TEST_3_-----");
18    lines = readLargeFileLines("K8B05a.txt");
19    for ( String line: lines ) System.out.println(line);
20    // write lines in large file with buffer
21    writeLargeFileLines("K8B05b.txt", lines);
22 }
```

Miscellaneous

- 'for each' Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- Regular Expressions

Enumerations (enum) In the simple case, **enum** defines a type as a set of constants. Every constant is represented only by its name.

Type definition:

```
enum Weekdays { MONDAY, TUESDAY, WEDNESDAY,  
                  THURSDAY, FRIDAY, SATURDAY, SUNDAY };
```

Variable definition:

```
Weekdays aDay;
```

Assignment:

```
aDay = Weekdays.TUESDAY;
```

In fact, **enum** defines a class that can also have constructors, variables and methods, in addition to constants.

- **enum** types are implicitly final
- **enum** constants are implicitly static

```
1 public class K8B06_EnumTage {
2     private enum Weekdays { MONDAY, TUESDAY, WEDNESDAY,
3                             THURSDAY, FRIDAY, SATURDAY, SUNDAY };
4
5     public static void main( String args[] ) {
6
7         Weekdays aDay = Weekdays.TUESDAY;
8         System.out.println("single day: " + aDay);
9
10        // values() returns array with all constants of this enum type
11        System.out.println("\nlist of all weekdays:");
12        for ( Weekdays day : Weekdays.values() )
13            System.out.println(day);
14    }
15 }
```

enum types can be translated like normal classes, or as nested static classes:

```
1 import java.util.EnumSet;
2
3 public class K8B07E_EnumBook {
4     public static enum Book {
5         JHTP8( "Java_How_to_Program_8e", "2015" ),
6         CHTP8( "C_How_to_Program_8e", "2016" ),
7         CPPHTP9( "C++_How_to_Program_9e", "2016" ),
8         CSHARPHTP5( "C#_How_to_Program_5e", "2012" );
9
10        //(constants are objects of their own class!)
11
12        private final String title;
13        private final String copyrightYear;
14
15        Book( String bookTitle, String year ) {
16            title = bookTitle;
17            copyrightYear = year;
18        }
19
20        public String getTitle() { return title; }
21        public String getCopyrightYear() { return copyrightYear; }
22    }
```

```
1  public static void main( String args[] ) {
2      System.out.println("All_books:");
3
4      for ( Book book : Book.values() )
5          System.out.println(book + ",_" +
6              book.getTitle() + ",_" + book.getCopyrightYear());
7
8      System.out.println("range selection:");
9
10     for ( Book book : EnumSet.range(Book.JHTP8,Book.CPPHTP9 ) )
11         System.out.println(book + ",_" +
12             book.getTitle() + ",_" +
13             book.getCopyrightYear());
14     }
15 }
```

Miscellaneous

- 'for each' Loops
- File I/O
- Enumeration types
- **Methods with parameter lists of variable length**
- Java Archives
- Regular Expressions

Methods with parameter lists of variable length:

- variable number of parameters of the same type
- implemented by combination of same-type parameters into an array

```
1 import java.util.Arrays;
2 class K8B08E_VarArg {
3
4     public static void sum (int ... numbers) {
5         int total = 0;
6         for (int i : numbers) total += i;
7         System.out.println("Sum of " +
8             Arrays.toString(numbers) +
9             ":\u2193" + total);
10        return;
11    }
12
13    public static void main (String args[]) {
14        int i1 = 5, i2 = 10, i3 = 15, i4 = 20;
15        sum (i1, i2);
16        sum (i1, i2, i3);
17        sum (i1, i2, i3, i4);
18    }
19 }
```

Miscellaneous

- 'for each' Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- **Java Archives**
- Regular Expressions

- In general one uses **.jar** files (Java archives) instead of single classes; usually these archives are signed for security reasons.
- Example `Test` for creating a **.jar** file (requires Linux shell):
 - ▶ Three java files: `A.java`, `B.java`, `C.java`
 - ▶ Each with `main` method; `A.main()` should be started
 - ▶ **A.class** uses **B.class** and **C.class**

```
1 javac A.java
2
3 echo "Main-Class:A" > Test.manifest
4 jar cmvf Test.manifest Test.jar A.class B.class C.class
5
6 java -jar Test.jar
```

Miscellaneous

- 'for each' Loops
- File I/O
- Enumeration types
- Methods with parameter lists of variable length
- Java Archives
- **Regular Expressions**

Example: consider regular expression for syntactically correct US ZIP Code

$$\{a, \dots, z, A, \dots, Z\}^2 \circ (\epsilon \cup \{ " " \} \cup \{ - \}) \circ \{1, \dots, 9\} \circ \{0, \dots, 9\}^4$$

in Java syntax: `[a-zA-Z]{2}[[-]]{0,1}[1-9][0-9]{4}`

- at the beginning: exactly 2 letters

`[a-zA-Z]{2}`

(2 lowercase or uppercase letters)

- at the end: exactly 5 digits, the first digit not 0

`[1-9][0-9]{4}`

(1 digit not 0, 4 arbitrary digits)

- in-between: nothing, 1 space, or 1 hyphen

`[[-]]{0,1}`

(' ' or '-', 0 or 1 times)

public boolean matches (String regex) tests if the String matches the template defined by the regular expression **regex**.

```
1 public class K9B09E_ZipCode {
2
3     public static void main(String[] args) {
4
5         String regex = "[a-zA-Z]{2}[_[-]]{0,1}[1-9][0-9]{4}";
6
7         boolean ok = args[0].matches(regex);
8
9         System.out.println("Input is " +
10             (ok?"valid":"invalid"));
11         System.exit(ok?0:1);
12     }
13 }
```

The exit code can be processed in a 'shell',
e.g. in a Unix bash:

```
java K9B09E_ZipCode Ab12345 && echo so it continues...
```

Java expressions for regular expressions

[abc]	1 character (a, b or c)
[^abc]	1 character (not a, not b and not c)
[a-dk-o]	1 character (in 'a to d' or in 'k to o')
.	exactly 1 arbitrary character

+	1 or more characters
*	0 or more characters
?	0 or 1 characters

{n}	exactly n characters
{n, }	at least n characters
{n, m}	between n and m characters

$[abc[ghi]]$	union (corresponds to $[abcghi]$)
$[a-z \& \& [def]]$	intersection (corresponds to $[def]$)
$[a-z \& \& [^\wedge def]]$	difference (corresponds to $[a-cg-z]$)

$[0-9]^+$	at least 1 digit
$[0-9]^*$	arbitrary many digits
$.^?$	0 or 1 arbitrary characters

$[a-z]\{n\}$	<i>n</i> lowercase letters
$a\{n, \}$	at least <i>n</i> times 'a'
$(ab)\{n, m\}$	between <i>n</i> and <i>m</i> times 'ab'

Predefined character classes:

- `\d` — a digit: `[0-9]`
- `\D` — a 'non-digit' character: `[^0-9]`
- `\s` — a 'whitespace' character: `[\t\n\x0B\f\r]`
- `\S` — a 'non-whitespace' character: `[^\s]`
- `\w` — a word character: `[a-zA-Z_0-9]`
- `\W` — a 'non-word' character: `[^\w]`

Usage of escape sequences for example with

`String REGEX = "\\d";` for a single digit

a small selection of 'boundary matchers':

- `^` — beginning of a line (for `\n` in a String)
- `$` — end of a line) `\n` in a String)
- `\b` — word boundary
- `\A` — beginning of the input
- `\Z` — end of the input (ignoring the final terminator `\n`, `\r`)
- `\z` — end of input

Examples for regular expressions:

regex	String	match?
foo	foo	+
cat.	cats	+
[rcb]at	cat	+
[rcb]at	hat	-
[^bcr]at	cat	-
[^bcr]at	hat	+
[a-c]	b	+
[a-c]	d	-
foo[1-5]	foo5	+
foo[^1-5]	foo6	+
[0-4[6-8]]	6	+
[0-4[6-8]]	5	-
[0-9&&[345]]	3	+
[2-8&&[4-6]]	3	-
[0-9&&[^345]]	6	+

regex	String	match?
.	@	+
a?		+
a?	a	+
a?	aa	-
a+	aaa	+
a+		-
a*		+
a*	aaaaaa	+
a{3}	aaa	+
a{3}	aaaa	-
a{3,}	aaaa	+
(hi){2}	hihi	+
.*hi	hellohi	+
\d+	12345	+
\D+	12345	-
\w+	hi_ho	+
\s		+

We can also determine parts of a String where a given regular expression matches:

```
1 public class RegexpTest {  
2  
3     public static void main(String[] args) {  
4  
5         String regex = "[a-zA-Z]{2}";  
6  
7         Matcher m =Pattern.compile(regex).matcher(args[0]);  
8  
9         while (m.find())  
10            {  
11                System.out.println("match:_" +m.group());  
12            }  
13     }  
14 }
```

regex	String	match found?
ing	singer	+
ing\b	singer	-
ing\b	sing	+
ing\B	singer	+
ing\B	sing	-
\s	_ _test	+
\s	test	-
\s	test_ _	+
^\s	_ _test	+
^\s	test	-
^\s	test_ _	-
\s\$	_ _test	-
\s\$	test	-
\s\$	test_ _	+