

Java overview

(annett.thuering@informatik.uni-halle.de / 2019)



Structure of a Java program

```
public class Classname {  
    public static void main(String[] args) {  
        // program code  
    }  
}
```

Compiler

javac *Classname*.java translate program to bytecode
java *Classname* VM runs bytecode

⚠ Filename has to equal class name

Comments

from // to end of line
from /* to */ whole block, can include multiple lines

Output

System.out.println(output); with line break
System.out.print(output); without line break
e.g. output: "text1" + variable + "text2"

Primitive data types

- integrated in language
- size, value range and operations are predefined

name	Size in bytes	Value range
byte	1 (8 bit)	$-2^7 - 2^7 - 1$
short	2 (16 bit)	$-2^{15} - 2^{15} - 1$
int	4 (32 bit)	$-2^{31} - 2^{31} - 1$
long	8 (64 bit)	$-2^{63} - 2^{63} - 1$
float	4 (32 bit)	$\pm 3.4 \cdot 10^{38}$
double	8 (64 bit)	$\pm 1.79 \cdot 10^{308}$
char	2 (16 bit)	Unicode
boolean	1 (8 bit)	true, false

Automatic type conversion

byte → short → int → long → float → double
char ↗

Constants

final data type CONSTANT = value;
e.g. final int YEAR = 2019;

Rules for identifiers

- name of variable, method, data type
- sequence of letters and numbers
- start with a letter
- \$ and _ count as letters
- differentiation between capital and lower case letters
- reserved words are not allowed

Declaration of variables

DataType *variableName*;

e.g. int number;
 int a, b, c;

Declaration (definition) with initialization

int number = 5;
String name = "Rudi";

Reading data

The class Scanner

```
import java.util.Scanner;  
public class Classname {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);
```

reading integers (return type int)

e.g. iNumber = input.nextInt();

reading rational numbers (return type double)

e.g. dNumber = input.nextDouble();

reading character strings (return type String)

e.g. word = input.next();

reading lines, blank spaces included (return type String)

e.g. line = input.nextLine();

The class BufferedReader

```
import java.io.*;  
public class Classname {  
    public static void main(String[] args)  
        throws IOException {  
        BufferedReader input = new BufferedReader  
            (new InputStreamReader(System.in));
```

reading lines, blank spaces included (return type String)

e.g. line = input.readLine();

Converting character strings (String) into numbers

convert String into whole number (int iNumber)

e.g. iNumber = Integer.parseInt(line);

convert String into fractioned number (double dNumber)

e.g. dNumber = Double.parseDouble(line);

Operators

Associativity (Compounding)

right-associative: prefix operator, conditional expression,
assignment operator

left-associative: all other operators

Priorities

pri	operators	comment
1	o.a, i++, a[], f(), .	Postfix operators
2	-x, !, ~, ++i	Prefix operators
3	new C(), (type) x	Object generation, Cast
4	*, /, %	Multiplication, etc.
5	+, -	Addition, Subtraction
6	<<, >>, >>>	Shift
7	<, >, <=, >=	smaller, larger, etc.
8	==, !=	equal, different
9	&	Bit by bit AND
10	^	Bit by bit XOR
11		Bit by bit OR
12	&&	logical AND
13		logical OR
14	?:	conditional expression
15	** Fehlerhafter Ausdruck **	assignment

Control structures

Conditional instructions (if ... else ...)

```
if (condition) {  
    // instructions are run,  
    // if condition true  
}  
else {  
    // instructions are run,  
    // if condition false  
}
```

Case differentiation (selection)

```
switch (variable) {
    case Value1: // This code is run if
                // variable == Value1
                break;
    case Value2: // This code is run if
                // variable == Value2
                break;
    ...
    default:    // This code is run, if
                // variable does not
                // contain any of the values
                // available for selection
}
```

⚠ data type of variable has to be primitive, String or a Wrapper Class.

Value1, Value2, ... are **constants**!

Repetitions

while – loop (as long as ...)

```
while (condition) {
    // instructions that are run,
    // as long as the condition is true
}
```

do while – loop (at least once – but as long as ...)

```
do {
    // instructions are run at least once
    // then as long as, condition is true
} while (condition);
```

for – loop (special case: counting loop)

```
for (initialization; condition; step by step) {
    // instructions
}
```

example: loop is run 10 times

```
for (int i = 0; i < 10; i++) {
    // instructions
}
```

for each – loop for arrays

```
double[] a = new double[10];
```

```
for (double x : a) {
    System.out.println(x);
}
```

Premature termination of loop

continue

- jumps to end of loop

break (termination)

- jump to first instruction after loop / switch

Exceptions

- ArrayIndexOutOfBoundsException (invalid index)

- StringIndexOutOfBoundsException (invalid position)

- NullPointerException (accessing null – reference)

- NumberFormatException (String does not contain numbers)

- ArithmeticException (e.g. division by 0)

Handling of exceptions:

```
try {
    // instructions
}
catch (exception1) {
    // instructions for error handling
}
catch (exception2) {
    // instructions for error handling
}
...
finally {
    // always run, if existing
}
```

Overloading methods

- overloaded methods = methods with same name but **different parameters** inside a single class

1. variant: **different number of parameters**

2. variant: **at least one parameter has a different data type**

- use case: semantically similar actions for different data types (look method System.out.println(...))

```
public void println()
public void println(String s)
public void println(int x)
...
```

Reference types

Arrays

one dimensional Arrays

characteristics: - elements stored in memory in succession
- elements from same data type

```
// declaration of reference variable
int [] intArray;
// reserving memory space for array
intArray = new int[length];
```

or

```
int [] intArray = new int[length];

// declaration + initialization
int [] even = { 2, 4, 6, 8, 10 };
```

length of an array

```
intArray.length
```

accessing element in array

- access over index

- smallest index: **0**; largest index: **length-1**

e.g. intArray[index] = 7;

multi dimensional arrays

⚠ In Java, a two dimensional array is actually a one dimensional array containing references to one dimensional arrays as elements itself.

- number of bracket pairs determines dimensionality

```
int [][] matrix; // declaration 2 dim.
int [][][] cuboid; // declaration 3 dim.
```

```
matrix = new int[5][10]; // 2 dim.
cuboid = new double[5][10][2]; // 3 dim.
```

length of the array

```
matrix.length // size of 1st dimension
matrix[2].length // length of 2nd line
                of 2nd dimension
```

accessing an element in an array

```
matrix[0][0] = 7; // 2 dim
cuboid[0][0][0] = 7; // 3 dim
```

Command line parameters

- given to program as it starts

- available via the array args of the main method

```
java ProgramName param1 param2 ...
```

```
public static void main(String[] args) {
    System.out.println(args[0]); → param1
    System.out.println(args[1]); → param2
    ...
}
```

Concepts of object oriented programming

1. concept: encapsulation of data and its processing functions (classes)

Structure of a class

```
class NameOfClass {  
    // attributes (variables/characteristics/  
    // instance variables)  
    // methods (processing of attributes)  
}
```

class is a data type (specifies the structure of the memory for an instance (object))

class = is one storage template for many objects!

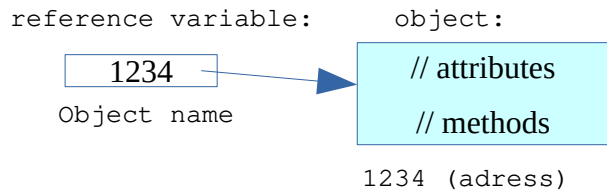
Objects / instances of a class

declaring reference variables

```
NameOfClass objectName;
```

creating object and assigning reference variable

```
objectName = new NameOfClass();
```



Operator **new**: reserving memory space

Methods

- mainly for communication with object (e.g. values are handed over to object [changing state of object] or attributes are read out from object [checking state of object]) → determining behaviour

```
return type MethodName(parameter list){  
    // instructions  
  
    // return of value if needed  
    // return expression;  
}
```

method does not **return value**

→ return value = **void**

return-instruction evaluates expression, terminates the method and returns value of expression

accessing components of an object

- access through operator **.**

```
objectName.Attribute      respectively  
objectName.Method(Parameter list)
```

The class Object

- superclass for all classes of a program

- contains the following important methods:

creating copy of object

```
protected Object clone()
```

comparing objects

```
boolean equals(Object o)
```

```
@Override  
public boolean equals (Object o) {  
    if (o == null) {return false;}  
    if (! (o instanceof NameOfClass))  
        {return false;}  
    if (o == this) {return true;}  
  
    // type conversion from o with (cast)  
    NameOfClass onew = (NameOfClass) o;  
  
    // own comparison with onew  
    ...  
}
```

returns object as String

```
String toString()
```

```
@Override  
public String toString() {  
    return "String";  
}
```

Constructors

- special methods

- serve to initialize an object (its attributes and reservation of memory)

- are run immediately upon creation of object

- a class can contain multiple constructors

characteristics:

1. no return value
2. same name as class

```
// constructor  
NameOfClass (parameter list) {  
    ...  
}
```

Access privileges

- are defined explicitly (private, public) / or implicitly (package default) **in front of** each attribute

private

- allows access **only** for objects of this class

package default

- allows access for all objects of same package

public

- allows access for all objects of any package

this

- contains reference for object itself

through it, all components of the object can be accessed

```
this.Attribute      respectively  
this.Method(parameter list)
```

Class attributes/ class methods (static)

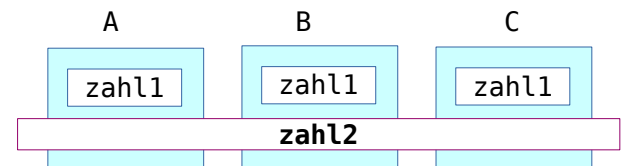
- exist only once for all objects of the same class

- exist even, if no object of class exists

! static methods can only access other static components!

```
class NameOfClass {  
    int zahl1;  
    static int zahl2;  
    ...  
}
```

```
A = new NameOfClass();  
B = new NameOfClass();  
C = new NameOfClass();
```



access possibilities to shared class attributes:

A.zahl2 B.zahl2 C.zahl2

NameOfClass.zahl2

Inheriting/ extending

2. concept: transferring attributes and methods to another class

Access privileges - continuation

protected

- allows access for objects of same package and for objects of sub classes from other packages

Superclass

```
class Superclass {
```

private components

protected components

package default components

public components

```
}
```

Subclass

A subclass extends its superclass by additional attributes and methods.

```
class Subclass extends Superclass {
```

Additional attributes and methods

access on components of superclass

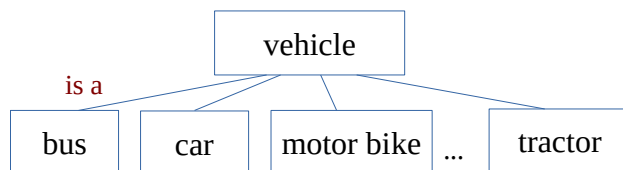
protected components

package default components

public components

```
}
```

Example: implementation of hierarchies



A bus is a vehicle.

Each instance of a subclass is an instance of a superclass as well.
`vehicle v1 = new car();` // polymorphism

Constructors

- constructors of a superclass are **not visible in the subclass**, even though they are contained and `public`
- constructors of the superclass will always be called upon
- succession: superclass constructors first, then other constructors further down in the hierarchy
- explicit call of a superclass constructor through:
`super(parameter list);`
- call of a superclass constructor is always the first instruction of a constructors, in sub classes

Reference **super**

- allows access to inherited components of directly superior superclass, in a sub class
- therefore allows **access to constructors**, **overwritten methods** and **hidden attributes**

Hidden attributes

- attributes in sub class that share the same name as attributes in superclass, hide inherited attributes of superclass **in methods of sub class**

```
class B {
    public int a;
    public B() {a = 5;}
}
class S extends B {
    public int a;
    public int a_ofSuper;
    public S() {
        a = 10;
        a_ofSuper = super.a;
    }
}
```

```
S testS = new S();
System.out.println(testS.a);           → 10
System.out.println(testS.a_ofSuper);   → 5
```

// polymorphism

```
B testBS = new S();
System.out.println("\n " + testBS.a); → 5
```

final - modifier

- methods defined as `final` can not be overwritten in inheriting sub classes
- classes defined as `final` can not be inherited

Overwriting of methods

- use case: customizing a method for the requirements of a sub class
- method has to have same signature (same name, list of parameters) and a compatible return value
- overwritten methods from superclass are only accessible through methods in sub class and only through the explicit statement `super`

```
class BM {
    public int zb;
    public BM() {zb = 2;}
    @Override public String toString () {
        return "zb = " + zb; }
}
class SM extends BM {
    public int zs;
    public SM() {zs = 20;}
    @Override public String toString () {
        return super.toString() +
            "\n zs = " + zs; }
}
```

```
SM testSM = new SM();
```

```
System.out.println(testSM);
```

zb = 2
zb =20

// polymorphism

```
BM testBMSM = new SM();
System.out.println(testBMSM);
```

abstract superclasses

- for defining fundamental properties in superclasses
- it is not possible to create an instance of an abstract class
- can contain **abstract methods = methods without implementation**

```
abstract class Asuper {
    int z;
    abstract public void print (int a);
    public int get_z () {return z;}
}
```

- class with abstract methods has to be declared `abstract`
- abstract methods can not be private

Generic classes

- template for a multitude of classes
- classes of this multitude are only different in the data type of some of their attributes (generic attributes)

```
class A {  
    int att;  
    ...  
}  
  
class B {  
    String att;  
    ...  
}  
  
class C {  
    Person att;  
    ...  
}
```

create instance:

```
A aobj = new A();  
B bobj = new B();  
C cobj = new C();
```

- a placeholder can be used in place of a data type

```
class<placeholder> ABCgen {  
    placeholder att;  
    ...  
    public placeholder getAtt () {return att;}  
    public void setAtt (placeholder w) {  
        att = w; }  
    @Override public boolean equals(Object o){  
        if(o instanceof ABCgen<?>) {  
            ABCgen<?> p = (ABCgen<?>) o;  
            ...  
        }  
    }  
}
```

- generic classes can only provide limited functionality to generic attributes
- when creating an instance of a generic class, a data type has to be specified, to replace the placeholder
- data type has to be a reference type

```
ABCgen<Integer> aobj = new ABCgen<Integer>();  
ABCgen<String> bobj = new ABCgen<String>();  
ABCgen<Person> cobj = new ABCgen<Person>();
```

ArrayList

- saves elements in an internal array
- size modifiable, contrary to array

```
import java.util.ArrayList;  
...  
ArrayList<DataType> aliste;  
aliste = new ArrayList<DataType>();
```

Methods in an ArrayList

```
return element at a specific position  
alist.get(Position)  
  
add element to end of list  
alist.add(Element)  
  
add element at a specific position  
alist.add(Position, Element)  
  
add all elements from a different list  
alist.addAll(Position, list2)  
  
change element at a specific position  
alist.set(Position, Element)  
  
delete element / clear list  
alist.remove(Element) / alist.clear()  
  
delete element at a specific position  
alist.remove(Position)  
  
return size of list  
alist.size()  
  
check, if a specific element is contained in list (returns true / false)  
alist.contains(Element)  
  
returns index of a specific element  
alist.indexOf(Element)
```

Interfaces

- dictates, what methods a class has to implement
- methods are always public and abstract (without keywords public, abstract)
- declared attributes are implicitly static, final and public and have to be initialized (→ constants)
- can be inherited by other interfaces
- enable multiple inheritance

sub classes can only extend one superclass, but implement multiple interfaces!

Example of a generic interface:

comparison of two objects:

```
interface Comparable<T> {  
    int compareTo (T obj2);  
}
```

return value: 0 if equal
 < 0 if object itself is smaller than obj2
 > 0 if object itself is larger than obj2

```
class Kl implements Comparable<Kl>{  
    private int wert;  
    ...  
    @Override  
    public int compareTo(Kl obj2) {  
        ...  
    }  
}
```

use, e.g. sorting:

```
import java.util.Collections;  
...  
ArrayList<Kl> kllist;  
kllist = new ArrayList<Kl>();  
...  
Collections.sort(kllist);  
...
```

Classes for Strings

String

 **Content can not be changed!**
(replace, toLowerCase ... **create new Strings**)

```
String satz = "Viel Spaß!";
```

```
accessing characters / length of string  
char charAt(int index) / int length()  
  
returns 1st position of ch (character) / st (String) in String  
int indexOf(int ch) / indexOf(String st)  
  
true, if String st is contained in String  
boolean contains(String st)  
  
comparison of two Strings; true, if both equal  
boolean equals (Object anObject)  
  
returns a part of String (without character at position end)  
String substring(int begin, int end)  
  
replacement of a character  
String replace(char old, char new)  
  
conversion to lower case / upper case letters  
String toLowerCase() / toUpperCase()
```

StringBuilder / StringBuffer

- modifiable Strings

```
import java.lang.StringBuilder;  
...  
StringBuilder satz;  
satz = new StringBuilder("Viel Spaß!");
```