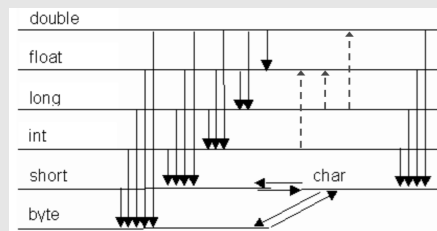


Typ Konversion



schwarze Pfeile: explizit
rote Pfeile: impliziet, evt Genauigkeitsverlust
alles andere impliziet

Switch-Statement

```
switch (Ausdruck) {
case Wert1:
    Anweisungen;
    break;
case Wert2:
    Anweisungen;
    break;
default:
    Anweisungen;
}
```

Wenn Ausdruck Wert1 entspricht, wird
Anweisung 1 ausgeführt...

Typ Polymorphismus

```
car extends vehicle
@Override
drive() {}
Vehicle v = new Car();
//calls drive of Car not Vehicle
v.drive();
```

Klasse hat eigenen Typ plus alle Typen der
Superklassen.
verschiedene "Brillen"
Dynamic Dispatch = dynamischer Typ zur
Laufzeit entscheidet über aufgerufene
Methoden.

Rekursion

```
public boolean groupSum(int start,
int[] nums, int target) {
    if (start >= nums.length) return
(target == 0);
    if (groupSum(start + 1, nums,
target - nums[start])) return
true;
    if (groupSum(start + 1, nums,
target)) return true;
    return false;}

Collection<String>
validParentheses(int nofPairs){
    Collection<String> list = new
ArrayList<>();
    if (nofPairs == 0)
{list.add("");} else {
    for (int k = 0; k < nofPairs;
k++) {
        Collection<String> infixes =
validParentheses(k);
        Collection<String> suffixes =
validParentheses(nofPairs - k -
1);
        for (String infix : infixes)
{
            for (String suffix :
suffixes) {
                list.add("(" + infix +
")" + suffix);}}}}
    return list;}
```

Generics

allg. Form

class Pair<T,U>

TypeBound (Bedingungen für Type)

class Node<T extends Comparable<T>>
Node<Person>//ok Node<Student>//Error

Multiple TypeBounds (mit & anhängen)

class Node<T extends Person &
Comparable<Person>>

generische Methode

Generics (cont)

```
public <T extends Person> T set (T element)
{...}
<T>: Bedingungen für Input
T: Rückgabotyp
```

Wildcard-Typ (Nutzen: Typargument nicht
relevant)

Node<?> undef --> kein Lesen & schreiben
ausser: Object o = undef.getValue();

Lambdas

```
(p1, p2) -> p1.getAge() -
p2.getAge()
```

```
p -> p.getAge() >= 18
```

```
people.sort((p1, p2) ->
p1.getName().compareTo(p2.getName()
));
```

```
people.sort(Comparator.comparing(P
erson::getLastName).thenComparing(P
erson::getFirstName));
```

```
Utils.removeAll(people, person ->
person.getAge() < 18);
```

```
people.sort(Comparator.comparing(p1
-> p1.getLastName().length()));
```

Syntax: (Parameter) -> {Body}

StreamAPI

```
people
.stream()
.filter(p -> p.getAge() >= 18)
.map(p -> p.getLastName())
.sorted()
.forEach(System.out::println);
people.stream().mapToInt(p ->
p.getAge())
.average().ifPresent(System.out::pr
intln);
Map<String, Integer>
totalAgeByCity =
```

StreamAPI (cont)

```
people.stream()
    .collect(
        Collectors.groupingBy(Person:
:getCity,
        Collectors.summingInt(Person:
:getAge));
```

endliche Quelle: `IntStream.range(1, 100)`

unendl. Quelle: `Random random = new Random();`
`Stream.generate(random::nextInt).limit(100)`

FileReader/Writer

```
private static void revertText()
throws FileNotFoundException,
IOException{
    try (FileReader reader = new
FileReader("input.txt");
        FileWriter writer = new
FileWriter("output.txt")) {
        int value = reader.read();
        String text = "";
        while (value >= 0) {
            text = (char)value + text;
            value = reader.read();
        }
        writer.write(text);
    }
```

Sichtbarkeit

public	alle Klassen
protected	Package und Sub-Klassen
private	nur innerhalb Klasse
(keines)	innerhalb Package

Datentypen

byte	8 bit (2 ⁷ bis 2 ⁷ -1)
short	16 bit
int	32 bit
long	64 bit (1L)
float	32 bit (0.1f)
double	64 bit

Operator-Prio

+, -, ++, --, ! (unär)

*, /, %

+, - (binär)

<, <=, >, >=

==, !=

&&

||

Rundungsfehler

0.1+0.1+0.1 != 0.3

Problem: `x == y`

double/float

Lösung: `Math.abs(x - y) < 1e-6`

Integer Literal

binär	0b10 = 2
oktal	010 = 8
hex	0x10 = 16
opt.	1000 = 1_000

Arithmetik

1 / 0 --> `ArithmeticException: / by zero`
1 / 0.0 --> `Infinity`
-1.0 / 0 --> `-Infinity`
0 / 0.0 --> `NaN`

Arithmetik Overloading

```
int operate(int x, int y) { ... }
double operate(int x, double y) {
    ... }
double operate(double x, double y) {
    ... }
```

`operate(1, 2);` --> meth 1
`operate(1.0, 2);` --> meth 3
`operate(1, 2.0);` --> meth 2

Overloading

```
class Graphic {
    void moveTo(Graphic other) // Method 1
}
class Circle extends Graphic {
    void moveTo(Graphic other) { // Method 2
    }
    void moveTo(Circle other) { // Method 3
    }
}
Graphic g = new Circle();
Circle c = new Circle();

g.moveTo(c) // Compiler: nur Methode 1 -> Laufzeit: Overriding durch Methode 2
c.moveTo(g) // Compiler: Overloading, nur Methode 2 passt für Argument g
c.moveTo(c) // Compiler: Overloading, Methode 3 spezifischer
g.moveTo(g) // Compiler: nur Methode 1 -> Laufzeit: Overriding durch Methode 2
```

Bedingungsoperator

`max = left > right ? left : right;`
wenn `left > right` wird `max = left`, sonst `max = right`

Package prio

own class (inc. nested class)
single type imports -> `import p2.A;`
class in same package
import on demand -> `import p2.*`



By **tarinya**
cheatography.com/tarinya/

Published 8th January, 2016.
Last updated 24th January, 2016.
Page 2 of 4.

Sponsored by **Readability-Score.com**
Measure your website readability!
<https://readability-score.com>

Enum

```
public enum Color {
    BLUE(1), RED(2);
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public int getColorValue() {
        return code;
    }
}
```

Methodenreferenz

```
people.sort(this::compareByAge)
int compareByAge(Person p1, Person
p2){return p1.age - p2.age;}

people.sort(Sorter::compareByAge)
static int compareByAge(Person p1,
Person p2){return p1.age - p2.age;}

Sorter sorter = new Sorter();
people.sort(sorter::compareByAge);
```

Serialisieren

```
OutputStream fos = new
FileOutputStream("serial.bin");
try (ObjectOutputStream stream =
new ObjectOutputStream(fos)){
    stream.writeObject(person);
}
```

needs Serializable interface
serialisiert alle direkt & indirekt referenz. Obj

Input/Output

```
try (BufferedInputStream
inputBuffer = new bis(new
FileInputStream(pathSource));
BufferedOutputStream outputB = new
bos(new
FileOutputStream(pathTarget))) {
    byte[] byteBuffer = new
byte[4096];
    int value =
inputBuffer.read(byteBuffer);
    while (value >= 0) {
        outputB.write(byteBuffer, 0,
value);
        value =
inputBuffer.read(byteBuffer);
    }
    catch (FileNotFoundException e) {
        System.out.println("File not
found: "+e);
    }
    catch (IOException e) {
        System.out.println("reading
Error: "+e);
    }
```

String Pooling

```
String a = "OO", b = "Prog", c =
"OOProg";
//true
a == "OO"; c == "OO" + "Prog";
(a+b).equals(c);
//false
a + b == c;
```

String Literals gepoolt. True Fälle -> es
werden keine neuen Objekte erstellt
Integer-Pooling -128 bis 127

Collections

```
Array int[] intArray = new int[3];

List List<T> al = new
ArrayList<>();
```

Collections (cont)

```
Set Set<T> hs = new
HashSet<>();

Map Map<U,T> hm = new
HashMap<>();

Iterator<T> it = a.iterator();
while(it.hasNext()) {...}

Collections.sort needs Comparable on
t(Object o) o

for (Map.Entry<Character, Double>
entry: map.entrySet())

LIST/SET: add(), remove()
LIST: get(index)
MAP: put(key, value), remove(key), getKey()
```

Interface vs. Abstract Class

Methoden	implizit public, abstract	evt abstract, nie private
Variablen	KONSTANTEN (impl. public static final)	normal, Constructor mit super()
Vererbung	implements	extends a a,b

Abstract Method (in abs. cl.):
public abstract void report(); kein
Rumpf{}
in interface: void report();
I. Methode mit Rumpf: default void
report() {...}
interface I1 extends I2,I3
class C2 extends C1 implements I1,
I2
wenn I1&I2 nun gleiche Methoden haben
(signatur) -> C2 muss ine der beiden
überschreiben. Zugriff möglich mit
I1.super.methode();
I2.super.methode();

Unchecked vs Checked Exceptions

Unchecked	Checked
Error	Try/catch oder Methodenkopf
RunTime-Exceptions	mögl. catch-Block: e.printStackTrace()

-> RunTime, NullPointerException, IllegalArgumentException, IndexOutOfBoundsException

eigene exception

```
class myE extends Exception { myE()
{}
MyE(String message) {
super(message);} }
```

Junit

@Before

```
public void setUp() {...}
```

```
@Test (timeout = 500, expected =
Exception.class)
```

```
public void testGoodName() {
assertEquals(expected, actual);
assertTrue(condition);}
```

@After -> tearDown()

EdgeCases testen (Grenzwerte, null,...)

equals

@Override

```
public boolean equals(Object obj) {
if(this == obj) {return true;}
if (obj == null) {return false;}
if (getClass() !=
obj.getClass()) {
return false;}
Student other = (Student)obj;
return regNumber ==
other.regNumber;}
```

HashCode überschreiben!

`x.equals(y) -> x.hashCode() == y.hashCode()

Grund: inkonsistenz bei Hashing. obj wird nicht gefunden, obwohl in Hash-Tabelle, x.contains(y) -> nutzt hashCode() & equals()

HashCode()

@Override

```
public int hashCode() {
return 31* firstName.hashCode()
+ 31 * lastName.hashCode();
}
```

Funktionsschnittstelle

@FunctionalInterface

```
interface Comparator<T> {
int compare(T first, T second);
}
```

genau eine Methode -> java.util.function

RegEx

Pattern pattern = Pattern.compile("reg")

vor ? optionaler Teil ([0]?[0-9]|2[0-3])

(){}*+?|\ als norm text *\(\ ...

Gruppennamen (?<NAME>)

Matcher matcher = pattern.matcher(string);

```
if (matcher.matches()){String one =
matcher.group("NAME")}
```



By **tarinya**

cheatography.com/tarinya/

Published 8th January, 2016.

Last updated 24th January, 2016.

Page 4 of 4.

Sponsored by **Readability-Score.com**

Measure your website readability!

<https://readability-score.com>