

Allgemein

- **var** ist eine Abkürzung damit der Classentyp nicht mehr eingegeben werden muss.
- Bei Array kommt die Klammer immer beim Datentyp: **int[]** arr.
- **length** bezieht sich auf die Grösse eines Arrays. **length()** bezieht sich auf die Anzahl Buchstaben in einem String.
- Um Strings miteinander zu verbinden, kann **+** verwendet werden.
- Typcast wandelt von einem tieferen Datentyp zu einem höheren um. Bsp. `int a = 2; double b = (double) a;`

Methoden

In Java müssen alle Methoden in einer Klasse definiert werden. Objektunabhängige Methoden mit **static** definieren, damit kein Objekt instanziiert werden muss, bevor die Methode gebraucht werden kann (gilt auch für Variablen bspw: `Math.PI`).

Aufruf über Klassenname.

Äquivalenzklassenbildung

boolean	Boolescher Wert	true, false
char	Textzeichen (UTF16)	'a', 'B', '0', 'ö', etc.
byte	Ganzzahl (8 Bit)	-128 bis 127
short	Ganzzahl (16 Bit)	-32'768 bis 32'767
int	Ganzzahl (32 Bit)	-2 ³¹ bis 2 ³¹ -1
long	Ganzzahl (64 Bit)	-2 ⁶³ bis 2 ⁶³ -1, 1L (L Suffix)
float	Gleitkommazahl (32 Bit)	0.1f, 2e4f (2 * 10 ⁴)
double	Gleitkommazahl (64 Bit)	0.1, 2e4

Wertebereich der Parameter in Bereiche zerlegen, die von der Funktion wahrscheinlich gleich behandelt werden.

Tests mit inneren und äusseren Randwerten erstellen.

Varargs (Variable Argumentlist)

```
public int sum(int... numbers)
{
    int total = 0;
    for(int el : numbers)
    {
        total += el;
    }
    return total;
}
```

Varargs funktionieren ähnlich wie Arrays.

Immer am Schluss der Parameterliste

Der Vorteil von Varargs ist das auch einzelne Elemente aus einem Array der Funktion übergeben werden kann.

Collections

Datensatz mit Elementen oder Schlüssel von Elementen.

Unterschied Set/ List: Im Set darf jedes Element nur einmal vorkommen. **Keine Dublikate!**

List	Folge von Elementen
Set	Menge von Elementen
Map	Abbildung Schlüssel → Werte

Unit test

Es gibt mehrere Arten von asserts die für Unit Test gebracht werden kann. Es ist wichtig das nur ein Unit Test nicht viel bringt, stattdessen sollten mindestens zwei bis drei Test für eine Funktion geschrieben werden. JUnit API's importieren.

assertEquals(expected, actual)

assertTrue(actual)

assertFalse(actual)

asserArrayEquals(expected, actual)

Float/Double: Delta muss vorhanden sein: `assertEquals(0.3, 0.1+0.2, 0.0001);`

```
@Test
void twoPlusTwoEuqualFour() // Generiert eine Testfunktion
{
    var calculator = new SimpleCalculator(); // Erstellt ein Neuse Objekt
    assertEquals(4, calculator.add(2, 2)); // verifiziert 4 mit dem Output
}
```

```
@Test
void twoPlusTwoEuqualFour()
{
    SimpleCalculator calculator = new SimpleCalculator();
    assertTrue(calculator.add(2, 2) == 4); // vergleicht Output mit 4
}
```

Enum

```
enum Level
{
    LOW, MEDIUM, HIGH
}

public class Main
{
    public static void main(String[] args)
    {
        Level myVar = Level.MEDIUM;
        switch(myVar)
        {
            case LOW:
                System.out.println("Low Level");
                break;
            case MEDIUM:
                System.out.println("Medium Level");
                break;
            case HIGH:
                System.out.println("High Level");
                break;
        }
    }
}
```

Eine Enum kann, genau wie eine Klasse, Attribute und Methoden haben. Unterschied ist, das Enum-Konstanten öffentlich, statisch und endgültig (unveränderbar - kann nicht überschrieben werden) sind.

Sie sind für alle Klassen im selben Package sichtbar.

Eine Enum nicht zur Erstellung von Objekten verwenden.

Anderen Klassen erweiterbar (kann aber Schnittstellen implementieren).

Wenn eine Methode im Enum vorhanden ist, dann muss bei der letzten Variable ein Semicolon geschrieben werden.

Enums können auch in einer Classe verwendet werden.

Verwendung

Wert bleiben konstant Bspw. Monatstage, Tage, Farben, Kartenspiele usw.

```
Medium Level
```

ArrayList

```
var friends = new ArrayList<String>();
friends.add("John"); // element am Ende hinzufügen
friends.add("Luke");
friends.add("Loris");
System.out.println(friends.get(1)); // element ausgeben
System.out.println(friends.size()); // größe der Liste
friends.set(0, "Hans"); // element ersetzen
friends.remove(1); // löscht Element 1
System.out.println(friends); // mögliche Ausgabe
friends.clear(); // entfernt alle Elemente
```

ArrayList ist im Grunde dasselbe wie ein Array, einfach besser und dynamischer. Es können Elemente hinzugefügt oder entfernt werden. ArrayList können nur Objekte enthalten.

Integer, String, Character, Long...

Wenn eine ArrayList ausgegeben werden soll -> Enhanced For

Loop

Enhanced For

```
String[] cars = {"Audi", "VW", "BMW"};
for(String vehicle : cars) // for(int i=0; i<cars.length; i++)
{
    System.out.println(vehicle); // System.out.println(cars[i]);
}
```

```
Audi
VW
BMW
```

Enhanced For loop funktionieren genau gleich wie normale for-Schleifen nur das man weniger code schreiben muss. Der Output ist bei beiden dasselbe: Audi, VW, BMW.

Vergleich von Objekten

```
public static void main(String[] args)
{
    String a = new String("abc");
    String b = new String("abc");
    System.out.println(a == b);
}
```

Dieses Beispiel ist ein Referenzvergleich. Somit werden nur die Adressen der beiden Objekte mit einander verglichen (**new** kreiert ein neues Objekt). Um den Inhalt zu Vergleichen muss mit der Equals Methode gearbeitet werden.

Equals

Wenn die Equals Methode nicht selbständig implementiert wird, so wird der Default der Equals Methode ausgeführt. Dieser vergleicht dann die Adresse der Objekten, ausser bei primitive Datentypen. Eine Ausnahme ist der String Datentyp und Wrapperklassen, solange dieser nicht von einem Objekt erzeugt wurde.

Die Equals Methode darf nicht für primitiven Datentypen verwendet werden!!

```
class Student //Ohne Vererbung
{
    String name; int studentId;
    ...
    public boolean equals(Object obj)
    {
        if(!(obj instanceof Student))
        {
            return false;
        }
        Student s = (Student)obj; //Typcast
        return Objects.equals(name, s.name)
        && studentId == s.studentId;
    }
}
```

```
class Student extends Person // mit Vererbung
{
    String name; int studentId;
    ...
    public boolean equals(Object obj)
    {
        if(obj == null || getClass() !=obj.getClass())
        {
            return false;
        }
        else
        Student s = (Student)obj;
        return Objects.equals(name, s.name) &&
        Objects.equals(studentId, s.studentId;
    }
}
```

instanceof überprüft ob das Objekt **obj** von einer Vererbung oder von einem Interface der Klasse Student ist.

Obj wird dann mit einem Typcast zu Student umgewandelt. Nun können alle Attribute verglichen werden.

Es können auf mehrere Arten die Equals Methode geschrieben werden. Am sichersten ist jedoch die im Code Beispiel. Siehe auf der letzten Seite für ein weiteres Beispiel.

Hashing

Idee: Schnelles finden von Objekten, da der Inhalt schon beschreibt wo das Element in einer Tabelle liegt.

HashSet: Finden eines Elements mit **contains** HashMap: Finden via Schlüssel (get).

Elemente werden gezielt durch eine Berechnung welche vom Inhalt des Elements abhängt an eine Position in der Hash-Tabelle verteilt.

Hash-Code

hashCode(); liefert Hash-Code für Objekt

Wichtig: Falls equals() überschrieben wird, muss auch hashCode() überschrieben werden.

x.equals(y) → x.hashCode() == y.hashCode(); Umkehrung muss nicht unbedingt gelten.

Eigene Hashfunktion

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

```
public static void main(String[] args)
{
    Set<String> carModels = new HashSet<>();
    carModels.add("Ferrari");
    carModels.add("Maserati");
    carModels.add("Lamborghini");
    boolean inserted = carModels.add("Ferrari"); // already present (no effect)
    System.out.println("Ferrari: inserted? " + inserted);
    System.out.println(carModels.contains("Volkswagen"));
    System.out.println(carModels);
}
```

Interface und Abstrakte Klassen

```
interface Vehicle
{
    void color(String c);
    void topSpeed();
    int wheelSize();
}

public class Car implements Vehicle
{
    String color;
    @Override
    public void color(String newColor)
    {
        color = newColor;
    }
    @Override
    public void topSpeed()
    {
        System.out.println("280km/h");
    }
    @Override
    public int wheelSize()
    {
        return 24;
    }
}
```

```
abstract class Vehicle
{
    String color = "blue";
    public void topSpeed()
    {
        System.out.println("300km/h");
    }
    abstract int wheelSize();
}

public class Car extends Vehicle
{
    String color = "yellow";
    public int wheelSize()
    {
        return 18;
    }
}
```

Interfaces sind Verträge zwischen Klassen und der Aussenwelt. Es wird nur der Methodenkopf geschrieben.

Alle Methoden des Interfaces müssen zwingend in der implementierten Klasse deklariert werden.

Das Interface kann auch Attribute enthalten.

Bsp: `int MAX_SPEED = 30;`

Das ist jedoch nicht so geschickt, da diese final sind. Besser wäre Abstrakte Klasse.

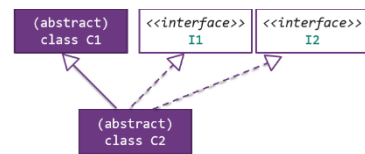
Vererbung

Bsp. `class car implements Vehicle, Brand`

Es können auch Interfaces miteinander gekoppelt werden.

Bsp. `interface Vehicle extends Brand`

Vorsicht ist geboten wenn verschiedene Implements dieselbe Methodennamen verwenden. Besser wäre Abstrakte Klasse.



`class C2 extends C1 implements I1, I2`

Abstrakte Klassen ist im Prinzip das Gleiche wie ein Interface. Der Unterschied ist jedoch das die Attribute nicht final sind. Somit könne sie für unterschiedliches gebracht werden. Auch Abstrakte Klassen können für die Mehrfachvererbung verwendet werden. Eine abstrakte Klasse kann nicht instanziiert werden.

Verwendung

Interface: Die Interfaces haben keine Beziehung zueinander.

Abstrakte Klassen: Können voneinander abhängig sein.

Wichtig auf einer bestimmten Klasse kann nur eine Abstrakte Klasse erweitert werden. Jedoch sind mehrere Implements möglich.

Set-Interface

Keine Duplikate! Gleichheitsprüfung mit equals().

```
Set<String> carModels = new HashSet<>();
carModels.add("Ferrari"); //Nochmal Ferrari geht nicht!
if(carModels.contains("Volkswagen")){ ... }
```

Map Interface

```
Map<Integer, Student> map = new HashMap<>();

Student st1 = new Student("Andrea", "Meier");
Student st2 = new Student("Bertha", "Müller");
Student st3 = new Student("Clara", "Schneider");

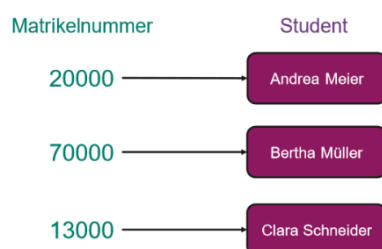
map.put(20000, st1);
map.put(70000, st2);
map.put(13000, st3);

Student st = map.get(13000);
for(Student s : map.values())
{
    system.out.println(s);
}
```

Map: Datenstruktur die für jeden Eintrag zwei Werte gespeichert hat.

Key – Value Value kann zBspw eine ArrayList sein

Mit `map.get(key)` kann der Value ausgelesen werden. Jeden Key gib es nur einmal!



Generics

```
public class Main
{
    public static void main(String[] args)
    {
        var printer = new Pinter<Integer>(23);
        printer.print();
    }
}
```

Generics ist dasselbe wie Templates im C++. Wenn eine Klasse oder Funktion mittels verschiedenen Datentypen gebraucht werden kann, so wird Generics eingesetzt. Das führt zu Code Verringerung. In der Main Klasse wird definiert welche Art von Datentyp verwendet werden soll.

```
public class Printer <T>
{
    T data; // Attribut
    public Printer(T data) // Konstruktor
    {
        this.data = data;
    }
    public void print()
    {
        System.out.println(data);
    }
}
```

Es können auch unterschiedliche Datentypen übergeben werden. Dabei kann die Klammer erweitert werden <T, D> und in der Main Klasse die einzelnen Datentypen definiert werden.

Wenn die Printer Klasse auch von einer selbst erstellten Datentyp (Klasse) verwenden möchte, so kann mit extends erweitert werden. Bsp. <T extends ImaginäreZahlen>

Generics im interface

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    // ...
}

public class Stack<T> implements Iterable<T> {
    // ...
    @Override
    public Iterator<T> iterator() {
        return elements.iterator();
    }
}
```

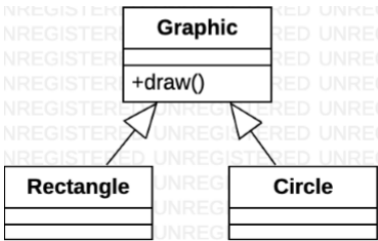
Typ Bounds von Parameter

Wenn nur T stünde, könnte ein Objekt übergeben werden, dass keine draw() Methode hat.

```
class GraphicStack<T extends Graphic> extends Stack<T> {
    public void drawAll() {
        for (T item : this) {
            item.draw();
        }
    }
}
```

Typen T müssen von Graphic erben

draw() nun definiert



Methoden Implementation	Methoden Aufruf
<pre>public <E> Stack<E> multiPush(E value, int times) { var result = new Stack<E>(); for (int i = 0; i < times; i++) { result.push(value); } return result; }</pre>	<pre>Stack<String> stack1 = multiPush("Hello", 100); Stack<Double> stack2 = multiPush(3.141, 3);</pre> Typ wird automatisch anhand Methodenargument erkannt.

Exceptions

```
try
{
    int[] myNumber = {1, 2,3};
    System.out.println(myNumber[10]);
}
catch(Exception e) //Typ von Exception bestimmen
{
    System.out.println("Something went wrong");
}
finally
{
    System.out.println("Try catch finished");
}
```

Damit ein Programm mit direkt zusammenbricht, werden Exceptions gebraucht. Dadurch können gewisse Input /Code Fehler gefunden werden. Mit dem Try wird ein bestimmter Codeausschnitt auf Fehler geprüft. Kein Fehler heisst catch wird ignoriert.

Wenn ein Fehler eintritt, so wird der Rest vom Try ignoriert und das Catch wird ausgeführt.

Finally wird immer ausgeführt, egal ob Catch oder nicht!!

```
Something went wrong
Try catch finished
```

```
static void checkAge(int age)
{
    if(age < 18)
    {
        throw new ArithmeticException("Access denied");
    }
    else
    {
        System.out.println("Access granted");
    }
}
```

Mit der throw-Anweisung können Sie einen benutzerdefinierten Fehler erstellen.

Die throw-Anweisung wird zusammen mit einem Ausnahmetyp verwendet. In Java gibt es viele Ausnahmetypen: ArithmeticException, FileNotFoundException, usw.

```
Exception in thread "main" java.lang.ArithmeticException: Access denied
    at Main.checkAge(Main.java:XY)
    at Main.main(Main.java:XZ)
```

```
try
{}
catch(noStringException | ShortStringException e)
{}
// ist das Gleiche wie
try
{}
catch(noStringException e)
{}
catch(ShortStringException e)
{}
}
```

Checked Exceptions

- Exception, von denen sich das Programm erholen kann.
- throws-Deklaration notwendig
- Laufzeitfehler
- Beispiel: Zu kurzes Argument

Unchecked Exceptions

- Laufzeitfehler, von denen sich das Programm nicht erholen kann.
- Keine throws-Deklaration
- Beispiel: NullPointerException, IllegalArgumentException

Polymorphismus und Vererbung

```
public class Bicycle extends Vehicle
{
    private String breakTyp;
    private boolean suspension;
    Bicycle(String brand, int topSpeed,
            String breakTyp, boolean suspension)
    {
        super(brand, topSpeed);
        this.breakTyp = breakTyp;
        this.suspension = suspension;
    }
    public String getBreakTyp()
    {
        return breakTyp;
    }
    public boolean getSuspension()
    {
        return suspension;
    }
}
```

```
public class Vehicle
{
    private String brand;
    private int topSpeed;

    Vehicle(String brand, int topSpeed)
    {
        this.brand = brand;
        this.topSpeed = topSpeed;
    }

    public String getBrand()
    {
        return brand;
    }
    public int getTopSpeed()
    {
        return topSpeed;
    }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Bicycle bike = new Bicycle("Imholz", 50, "clutch", true);
        System.out.println(bike.getSuspension());
    }
}
```

Konstruktor

Konstruktor der Parentclass müssen im Konstruktor der Childclass immer als erstes aufgerufen werden.

Vererbung

Eine Subklasse erbt **alle** Instanzvariablen und Methoden von allen Superklassen. Bei einer Vererbung wird von einer «**Ist-Beziehung**» gesprochen. Bsp. Velo ist ein Fahrzeug.

Klassen können mit der Hilfe eines UML Diagramms grob dargestellt werden. Vererbung ist nur in einer Richtung gültig. Bicycle kann Eigenschaften von Vehicle übernehmen, aber nicht umgekehrt.

Mehrfachvererbung

Eine Klasse wird von mehr als einer Vaterklasse abgeleitet. Dies muss in Java mit der Hilfe von Interfaces oder Abstrakte Klasse umgesetzt werden.

Jede Klasse erbt von Object -> implizit, wird nicht angegeben. Bspw. Equals Methode.

Polymorphismus

Objekt kann auch anstelle von Oberklassen eingesetzt werden (hat auch Typ der Oberklasse). **Vererbungsrichtung anschauen**

Car c = new Car(); Vehicle v = new Car(); Object b = new Car();

Ein Car ist auch ein Car, ein Vehicle und ein Objekt.

Overriding

Subclass implementiert Methoden der Basisklasse neu. Gleiche Signatur -

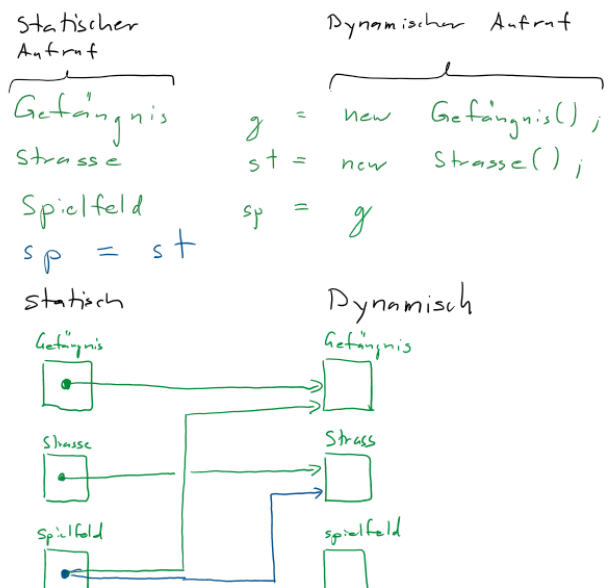
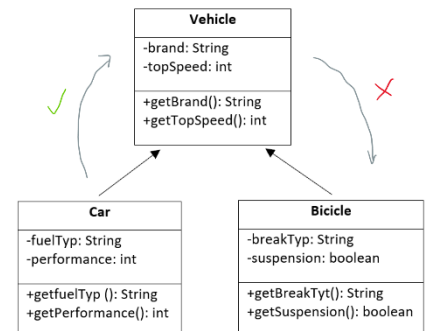
> gleicher Name und Parameterliste. Siehe Beispiel bei Interfaces.

Überschriebene Methoden mit **super.methode();** aufrufen.

Dynamic Binding

Statischer Typ: Gemäss Deklaration zur **Compile Zeit**.

Dynamischer Typ: **Effektiver Typ zur Laufzeit**. Super.methode();



Methoden Referenz / Lambda

Ein Lambda Ausdruck ist eine Abkürzung um Implementierung eines Functional Interfaces zu vereinfachen.

Functional Interface: Interface mit nur einer Methode, Annotation: `@FunctionalInterface`

```
// Funktioniert nicht da Lebewesen keine Klasse
Lebewesen Tier = new Lebewesen();

// Funktioniert ist aber aufwendig
Lebewesen Tier = new Lebewesen(){
@Override
public void foo(){
    System.out.println("Java");
}}

// Mit Lambdas
Lebewesen Tier = ()-> System.out.println("Java");
```

```
@FunctionalInterface
public interface Lebewesen
{
    void foo();
}
```

Syntax: (Parameter) -> { Methodenkörper }

Rückgabotyp wird inferiert (berechnet)

Parameterliste

```
(p1, p2) -> {
    return Integer.compare(p1.getAge(), p2.getAge());
}
```

Rückgabotyp
implizit int

Method Body

```
public class Demo02MethodRef
{
    void print(List<Person> people)
    {
        people.sort(this::compareByAge); // Sortierung der Personen
        System.out.println(people);
    }

    int compareByAge(Person p1, Person p2)
    {
        return Integer.compare(p1.getAge(), p2.getAge());
    }

    public static void main(String[] args)
    {
        new Demo02MethodRef().print(Samples.getPeopleList());
    }
}
```

```
public class Demo03LambdaShort
{
    public static void main(String[] args)
    {
        var people = Samples.getPeopleList();
        // ...
        people.sort((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));
        people.sort((p1, p2) -> p1.getLastName().compareTo(p2.getLastName()));
        System.out.println(people);
    }
}
```

Stream-API

```
public class StreamDemo01 {
    public static void main(String[] args) {
        var people = Samples.getPeopleList(); // neues Objekt erstellen
        people
            .stream()
            .filter(p -> p.getAge() >= 60) // filter nach Alter > 60
            .map(p -> p.getLastName())      // das gefilterte in map aufbewahren
            .sorted()                       // sortieren
            .forEach(System.out::println); // Liste wird ausgegeben
    }
}
```

```
public class Demo04Match {
    public static void main(String[] args) {
        var people = Samples.getPeopleList();

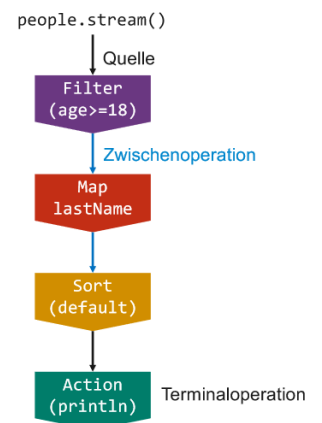
        boolean adultsOnly = people
            .stream()
            .allMatch(p -> p.getAge() >= 18);
        System.out.println("All adults: " + adultsOnly);

        boolean hasChildren = people .stream() .anyMatch(p -> p.getAge() < 18);
        System.out.println("Any children: " + hasChildren);
    }
}
```

Stream API wird dann eingesetzt, wenn eine Liste gefiltert oder angepasst werden soll. Zuerst wird ein neues Objekt erstellt, welches die Liste der Daten enthält. Danach kann mit `.stream()` den Stream gestartet und an der Liste herumgebastelt werden. Die neue Liste kann nun auch weiterverwendet werden.

People

```
.stream()
.filter(p -> p.getAge() >= 18)
.map(p -> p.getLastName())
.sorted()
.forEach(System.out::println);
```



File I/O

```
public class Demo03OutputTest
{
    public static void main(String[] args) throws IOException
    {
        try (var out = new FileOutputStream("test.bin", true))
        {
            for (int i = 0; i < 1000; i++) {
                out.write(i % 256);
            }
        }
    }
}
```

```
public class Demo08SimpleText
{
    public static void main(String[] args) throws IOException
    {
        Files
            .lines(Path.of("quotes.txt"), StandardCharsets.UTF_8)
            .forEach(System.out::println);

        var lines = Files
            .readAllLines(Path.of("quotes.txt"), StandardCharsets.UTF_8);
        Files
            .write(Path.of("copy.txt"), lines, StandardCharsets.UTF_8);
    }
}
```

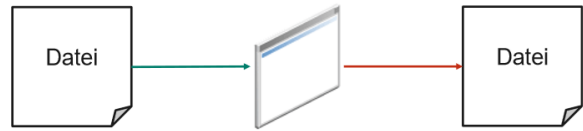
Input Output java.io.*

Externe Datenquellen/Datensenzen

Byte Streams

Standard Input/Output: System.in, System.out, System.err

FileInputStream FileOutputStream

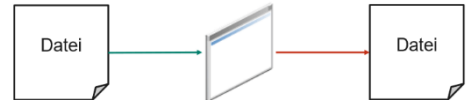


Input	Output
File-Reader <pre>try (var reader = new FileReader("quotes.txt")) { int value = reader.read(); while (value >= 0) { char c = (char)value; // use character value = reader.read(); } }</pre> FileReader: new FileReader(f) ⇔ new InputStreamReader(new FileInputStream(f)) Systemabhängiges Character Set -1 = end of file 16-bit char Wichtige Methoden int read(byte[] b, int offset, int length) • Lese length Bytes in Array b ab Index offset • Rückgabe = gelesene Bytes (-1 ⇔ end of stream) Datei binär einlesen: <code>byte[] data = Files.readAllBytes(Path.of("in.bin"));</code>	<pre>var out = new FileOutputStream("test.data"); while (...) { byte b = ...; out.write(b); } out.close();</pre> Datei neu anlegen bzw. überschreiben Schreiben («Flush») des Rests beim Schliessen An Datei anhängen, falls existiert <pre>new FileOutputStream("test.data", true)</pre> Wichtige Methoden: void write(byte[] b, int offset, int length) void flush() • Schreibt eventuell noch im Cache zwischen-gespeicherte Ausgaben • Implizit bei close() Datei binär schreiben: Files.write(Path.of("out.bin"), data);

Character Streams

Textweiser File-Input/Output.

FileReader FileWriter



Input	Output
File-Reader <pre>try (var reader = new FileReader("quotes.txt")) { int value = reader.read(); while (value >= 0) { char c = (char)value; // use character value = reader.read(); } }</pre> Systemabhängiges Character Set -1 = end of file 16-bit char Zeilenweises Lesen <pre>try (var reader = new BufferedReader(new FileReader("quotes.txt")) { String line = reader.readLine(); while (line != null) { System.out.println(line); line = reader.readLine(); } }</pre> BufferedReader Zeilenweiser Buffer FileReader Character Stream File Zeilenweises Lesen von InputStream <pre>try (var reader = new BufferedReader(new InputStreamReader(fileInputStream, Charset.forName("UTF-8")))) { String line = reader.readLine(); while (line != null) { System.out.println(line); line = reader.readLine(); } }</pre> BufferedReader Zeilenweiser Buffer InputStreamReader Character Stream FileInputStream Byte Stream File Ganze Text-Datei einlesen <code>List<String> lines = Files.readAllLines(Path.of("in.txt"), StandardCharsets.UTF_8);</code> Alle Zeilen als Stream API: <code>Stream<String> stream = Files.lines(Path.of("in.txt"), StandardCharsets.UTF_8);</code>	File-Writer <pre>try (var writer = new FileWriter("test.txt", true)) { writer.write("Hello!"); writer.write('\n'); }</pre> Append String schreiben Einzelnen char schreiben Ganze Text-Datei schreiben <code>Files.write(Path.of("out.txt"), lines, StandardCharsets.UTF_8);</code>

```

public static void main(String[] args)
{
    int[][] array = {{0,1,3},{2,3,5},{2,3,4}};
    var duplicate = printArray(array);
    duplicate[1][1] = 0;
}

public static int[][] printArray(int[][] a)
{
    int rows = a.length;
    int cols = a[0].length;

    for(int i = 0; i < rows; i++)
    {
        for(int k = 0; k < cols; k++)
        {
            System.out.print(a[i][k]);
            System.out.print(", ");
        }
        System.out.print("\n");
    }
    return a;
}

```

```

public class Main {

    public static void main(String[] args) throws IOException
    {
        numerize("Input.txt", "Output.txt");
    }

    public static void numerize(String in, String out) throws IOException{
        String zw = null;
        int counter = 1;
        try(var reader = new BufferedReader(new FileReader(in))){
            try(var writer = new FileWriter(out,true)){
                zw = reader.readLine();
                while(zw != null){
                    writer.write(counter + ":" + zw + "\n");
                    zw = reader.readLine();
                    counter++;
                }

                reader.close();
                writer.close();
            }
        }
        catch (IOException E){
            System.out.println(E);
        }
    }
}

```

Code Beispiele

```
public class Person implements Comparable<Person> {
    private final String firstName;
    private final String lastName;
    private final int age;
    private final String city;

    public Person(String firstName, String lastName, int age, String city) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.city = city;
    }

    public String getFirstName() {return firstName;}
    public String getLastName() {return lastName;}
    public int getAge() {return age;}
    public String getCity() {return city;}

    @Override
    public int hashCode() {
        return Objects.hash(age, city, firstName, lastName);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        Person other = (Person) obj;
        return age == other.age && Objects.equals(city, other.city) &&
            Objects.equals(firstName, other.firstName)
            && Objects.equals(lastName, other.lastName);
    }

    @Override
    public String toString() {
        return firstName + " " + lastName
            + " (" + age + ", " + city + ")\n";
    }

    @Override
    public int compareTo(Person other) {
        int value = lastName.compareTo(other.lastName);
        if (value == 0) {
            value = firstName.compareTo(other.firstName);
        }
        if (value == 0) {
            value = Integer.compare(age, other.age);
        }
        if (value == 0) {
            value = city.compareTo(other.city);
        }
        return value;
    }
}
```

```
public class Samples {
    public static List<Person> getPeopleList() {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Peter", "Vogt", 85, "Zürich"));
        people.add(new Person("Clara", "Meier", 66, "Rapperswil"));
        people.add(new Person("Andrea", "Schneider", 25, "Rapperswil"));
        return people;
    }
}
```