Java für CPP-Programmierer

HS 2024, Peter Bühler

Fabian Suter, Martina Knobel

31. Januar 2025 1.0.2

1 Basics

1.1 Grundsätzlich: Java ...

- kennt keine Zeiger / Pointer
- kennt keine Funktionen (reine OO-Sprache)
- kennt kein Überladen von Operatoren
- kennt keine Destruktoren (Garbage-Collector gibt Speicher frei)
- kennt keine Mehrfachvererbung
- stellt eine umfangreiche Klassenbibliothek zur Verfügung
- -Programme sind plattformunabhängig
- ist weniger hardwarenah wie C++

1.2 Java Runtime

Der Compiler erzeugt anders als in C++ **Bytecode**, welcher anschliessend auf einer **J**ava **V**irtual **M**achine laufen kann. ⇒ Java plattformunabhängig, da jede Plattform (MacOS) eine JVM hat.

2 Datentypen

Ein Datentyp besteht aus *Werten* und *Operationen*. In Java:
Primitive Datentypen ⇔ **Referenzdatentypen**.

2.1 Primitive Datentypen

Plattformunabhängig gleicher Wertebereich, keine `unsigned`-Typen.

Typ	Beschr.	Beispiele
boolean	Bool'scher Wert	true, false
char	Textzeichen (UTF16)	'a', 'B', 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
float	Gleitkommazahl (32 Bit)	0.1f, 2e4f
double	Gleitkommazahl (64 Bit)	0.1, 2e4

Gleitkommazahlen ohne Angaben sind automatisch `double`.

2.1.1 Überlauf / Unterlauf

Ganzzahlen	2147483647 + 1 → -2147483648
Gleitkommazahlen	2 * 1e308 → `POSITIVE_INFINITY` 5e-324 / 2 → 0.0

2.1.2 Text-Literale

char mit Apostrophen: `'A'`, `'\n'` (NewLine), `'\''` (Apostroph)
String mit Anführungszeichen: `"Say \"hello\"!\n"`

2.1.3 Undefinierte Operationen

Ganzzahlen werfen bei Division/Modulo durch 0 einen Fehler bzw. eine `null`-Referenz generiert eine Exception

Gleitkommazahlen werfen bei Division durch 0 ein `POSITIVE_INFINITY` resp. `NEGATIVE_INFINITY`

Bei undefinierten Rechnungen wie 0 / 0 wird `NaN` zurückgegeben.

2.2 Referenzdatentypen

2.2.1 null-Referenz

Spezielle Referenz auf "kein Objekt", vordefinierte Konstante, welche für alle Referenztypen gültig ist. *Dereferenzieren* der `null`-Referenz generiert eine `NullPointerException`

2.2.2 Klassen: Siehe Kapitel 3.1

2.2.3 Arrays

Arrays speichern wie in C++ mehrere Elemente mit selbem Datentyp. Der Zugriff erfolgt über Index, die Elemente liegen nebeneinander im Speicher. Die Grösse des Arrays muss bei der Deklaration festgelegt werden und ist später nicht mehr änderbar.
Die Anzahl der Elemente kann in Java direkt mit `length` abgerufen werden, siehe auch **12.3**.
Falls der Index ungültig ist, wird eine
ArrayIndexOutOfBoundsException geworfen.

2.2.4 Enums

Ähnliche Verwendung wie in C++. Enums können als Konstantenlisten verwendet werden:

```
public enum Weekday{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
     ;
    public boolean isWeekend(){
        return this == SATURDAY || this == SUNDAY;
    }
}
Weekday currentDay = ...
if (currentDay.isWeekend()){...}
```

2.2.5 Collections: Siehe auch Kapitel 4

Interface	Klassen	Eigenschaften
List	ArrayList, LinkedList	Folge von Elementen, Duplikate möglich
Set	HashSet, TreeSet	Menge von Elementen, keine Duplikate
Map	HashMap, TreeMap	Abbildung Schlüssel ↪ Werte, keine Duplikate

2.3 Typumwandlung

Implizit:
Klein zu Gross, kein Cast-Operator erf.

```
int a = 4711;

float b = a;

double c = b;
```

Explizit:
Gross zu Klein, Cast-Operator erf.

```
int y = 0x11223344;
short x = (short) y; // x = 0x3344

double w = 4.7;
int v = (int) w; // v = 4
```

3 Objektorientierung

3.1 Klassen und Objekte

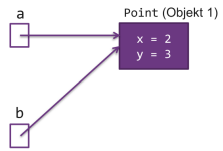
Klassen bestehen aus **Variablen** und **Methoden**.
Objekte können aus Klassen erzeugt werden: `Point a = new Point()`
Konstruktoren können in den meisten IDEs automatisch generiert werden. Sofern kein Konstruktor definiert ist, wird der **Default-Konstruktor** verwendet. Die Instanzvariablen werden mit Default-Werten initialisiert (primitive mit `0`, Referenzdatentypen mit `NULL`)

Klassen arbeiten mit Referenzen, dies muss bei Zuweisungen beachtet werden:

```
Point a = new Point();
a.x = 0;
a.y = 1;

Point b = a;
b.x = 2;
b.y = 3;

// Ausgabe: 2, 3
System.out.println( a.x + ", " + a.y);
```



3.2 Methoden

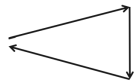
3.2.1 Parameter

Java verwendet *immer* **Call by Value**. Die Argumente werden kopiert und als Parameter übergeben. Dies kann zu unterschiedlichem Verhalten je nach Datentyp führen:

Primitive Datentypen:
Methode arbeitet mit Kopie des Wertes

```
int x = 5;
square(x);

// x bleibt 5
```

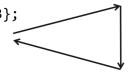


```
static void square(int x) {
    x = x * x;
    System.out.println(x);
}
```

Referenzdatentypen:
Methode arbeitet mit Kopie der Referenz

```
int[] a = {1, 2, 3};
square(a);

// a geändert
```



```
static void square(int[] a) {
    for (int i=0; i<a.length; ++i) {
        a[i] = a[i] * a[i];
    }
}
```

3.2.2 Variadische Methoden

Falls bei Funktionen die Anzahl der Argumente nicht im Voraus bekannt ist, kann folgende Syntax verwendet werden:
`static int sum(int... numbers){}`
Der Compiler generiert ein Array aus der Parameterliste.

3.3 Methodenreferenzen

Anstatt Hilfsklassen für häufig verwendete Methoden zu erstellen, kann auch mit Methodenreferenzen gearbeitet werden, wie in C++ mit Funktionszeigern. Für die Referenzierung braucht es eine Funktionsschnittstelle und die Implementierung. Methodenreferenzen benötigen eine aufrufende **höherwertige** Funktion.
Referenz-Varianten:

`this::compare`	Methode `compare` im selben Objekt
`other::compare`	Methode `compare` in Objekt *other*
`MyClass::compare`	Statische Methode in *MyClass*
`MyClass::new`	Konstruktor der Klasse *MyClass*

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

Funktionsschnittstelle

```
Comparator<Person> myComparison = this::compareByAge;
```

Methodenreferenz

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

Implementierung

3.3.1 Lambdas ~ 3 Zeilen → sonst: siehe auch 12.9

Ad-Hoc Implementierung anstatt einer expliziten Methodenreferenz.

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

`Person`,`{}` und `return` sind optional, wenn nur ein Ausdruck enthalten

3.4 Unit Testing: Siehe auch Kapitel 12.5

Unit Testing ist eine der Varianten, um Bugs zu verhindern. In guten Unit Tests sollen möglichst alle relevanten Fälle abgedeckt sein. Dazu gehören Standardfälle (im Bereich der Funktion, auch **Äquivalenzklassen** genannt) und Edge Cases (z.B. 0, max. und min. Bereich, usw.).

- Pro Klasse eine Test-Klasse
- Pro Testfall eine Methode

3.4.1 Assert-Methoden

Methode	Bedingung
assertEquals(expected, actual)	für prim. und Referenztypen
assertNotEquals(expected, actual)	
assertSame(expected, actual)	actual == expected
assertNotSame(expected, actual)	actual != expected
assertTrue(condition)	condition
assertFalse(condition)	!condition
assertNull(value)	value == null
assertNotNull(value)	value != null

```
package javac.v2.demo;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class Demo07AbsTest {

    @Test
    @DisplayName("Positive Number")
    void testPositiveValue() {
        long in = 23;
        long out = 23;
        assertEquals(out, Demo07Abs.abs(in));
    }
}
```

Optionaler Name für Anzeige

Prüfe, ob Resultat stimmt

3.4.2 Sichtbarkeit

Keyword	Sichtbar für
public	Alle Klassen
protected	Klassen im selben Package und abg. Klassen
(keines)	Klassen im selben Package
private	Nur eigene Klasse

Für Zugriffe auf `private`-Variablen können Getter- und Setter-Methoden definiert werden. Siehe auch 12.6

3.5 Generics → typ-unabhängige Frames erstellen

Nicht enthalten: Typebounds, Wildcardtypen, Covarianz, Type-Erasure

3.5.1 Generische Variablen

Die Variable dient als Platzhalter innerhalb einer generischen Klasse, Methode oder Interface. Sie wird mit `<>` umschlossen und kann innerhalb der Klasse,... wie ein normaler Typ verwendet werden.

Häufig verwendete Namen:

E	Element	T	Type
K	Key	V	Value
N	Number	S, U ,V,...	2ter, 3ter, 4ter Type

Mehrere Typ-Variablen gleichzeitig sind zulässig, z.B. `<T, U>`

3.5.2 Generische Klassen

Klasse mit Typ-Variable (Platzhalter für unbekannten Typ).
Verschachtelung bei der Anwendung ist zulässig.

Typ-Variable

```
class Stack<T> {
  // …
}
```

Typ-Argument

```
Stack<String>  stack1;
Stack<Integer> stack2;
```

Bei der Anwendung macht der Compiler eine statische Prüfung des Datentyps. der Type-Cast entfällt (auto-boxing, -unboxing)

3.5.3 Generische Interfaces

Gleiche Syntaxregeln wie bei generischen Klassen, jede Klasse kann generische Interfaces implementieren.

Anwendung:

```
public class Stack<T> implements Iterable<T> {

    private List<T> values = new LinkedList<>();

    public void push(T obj) { values.add(obj); }
    public T pop() { return values.removeLast(); }
    …

    @Override
    public Iterator<T> iterator() {
        return new StackIterator<T>(values);
    }
}
```

gleiche Typ-Variable T für Stack und Iterable

Implementiert Iterator<T>

Einsatz

```
for (var obj : stack) {
  System.out.println(obj);
}
```

```
public class StackIterator<T> implements Iterator<T> {
    private List<T> values;
    private int currentIndex;

    public StackIterator(List<T> values) {
        this.values = values;
        this.currentIndex = values.size() - 1;  // top of stack
    }

    @Override
    public boolean hasNext() { return currentIndex > -1; }

    @Override
    public T next() {
        T obj = values.get(currentIndex);
        --currentIndex;
        return obj;
    }
}
```

3.5.4 Generische Methoden

- unabhängig von generischen Klassen und Interfaces
- Implementation kann in normalen, nicht generischen Klassen erfolgen und ist auch in statischen Methoden zulässig.

Die Typ-Variable muss in Klammer `<>` vor dem Rückgabewert stehen.

```
public static <T> List<T> merge(List<T> a, List<T> b) { ... }
```

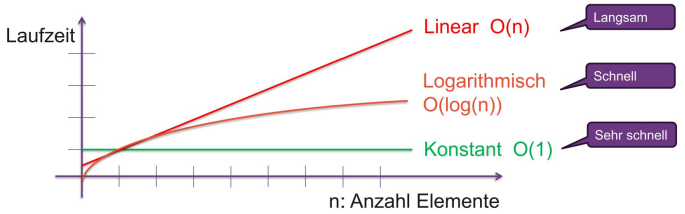
4 Collections → Nur Referenzdatentypen ablegbar

Beim Hinzufügen des Elements:
Objekt selber **nicht** kopiert, nur eine Referenz wird abgelegt.

Grundlegende Collections:
- **List** Folge von Elementen
- **Set** Menge von Elementen
- **Map** Abbildung Schlüssel → Werte

4.1 Asymptotisches Laufzeitverhalten

Laufzeit	Beschr.	Beispiele
O(1)	Konstant	Indexzugriff Array
O(log(n))	Logarithmisch	Binärsuche
O(n)	Linear	Lineare Suche
O(n*log(n))	LogLinear	Schnelle Sortierverfahren: QuickSort, MergeSort
$O(n^2)$	Quadr.	Einfache Sortierverfahren: SelectionSort, InsertionSort
$O(n^3)$	Kubisch	Matrizen-Multiplikation



4.2 Wrapper-Objekt

Um primitive Datentypen in Collections verwenden zu können, müssen sie verpackt (*Wrapping*) werden. Dies geschieht meist **implizit**, es muss nur beim Datentyp der Collection definiert werden.

- Für alle primitiven Datentypen gibt es eine Wrapperklasse

```
// Boxing
int x = 123;
Integer obj = Integer.valueOf(x);

// Unboxing
Integer obj = ..
int x = obj.intValue();
```

Primitiver Typ	Wrapper-Klasse
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

4.3 ArrayList

- geordnete Folge von Elementen mit demselben Referenzdatentyp
- Elemente können einfach hinzugefügt oder entfernt werden.
- Duplikate oder `null`-Einträge sind möglich
- Zugriff auf Elemente erfolgt über Index (0 ... size()-1).
- Die Liste verwendet intern ein Array zur Verwaltung der Elemente.
- Zu Beginn enthält sie, sofern nicht anders definiert, 10 Elemente
- Wird bei Erreichen der Kapazität mit Faktor 1.5 mulitpliziert.

```
ArrayList<String> module = new ArrayList<>();

module.add("OO");           // append at end
module.add("ICTh");
module.add(0, "Bsys1");     // insert at index 0

String x = module.get(1);   // get at index 1
module.set(0, "Bsys2");     // replace at index 0

module.remove("ICTh");      // remove element
module.remove(1);           // remove at index 1

int size = module.size();   // get length list
```

Type	Method / Description → s./spec. = specified, elem = element
void	`add(int index, E elem)` → Inserts s. elem at s. pos. in this list.
boolean	`add(E e)` → Appends the spec. elem to the end of this list.
boolean	`addAll(int index, Collection<? extends E> c)` → Inserts all elems in spec. collection into this list, starting at spec. pos.
boolean	`addAll(Collection<? extends E> c)` → Appends all elems in the spec. collection to the end of this list, in the order that they are returned by the spec. collection's Iterator.
void	`clear()` → Removes all elems from this list.
Object	`clone()` → Returns a shallow copy of this `ArrayList` instance.
boolean	`contains(Object o)` → Returns `true` if list contains spec. elem

4.3.1 ArrayList: Kosten

Operation	Methode	Effizienz
Index-Zugriff	get(), set()	Sehr schnell (direkter Zugriff)
Hinzufügen	add()	Langsam (umkopieren) Sehr schnell (ohne umkop.)
Entfernen	remove(int)	Langsam (umkopieren)
Finden	contains()	Langsam (durchsuchen)

4.4 LinkedList

Funktioniert ähnlich wie ArrayList. Die Implementierungerfolgt mit einer doppelt-verketteten (vor- und rückwärts) Liste. Es erfolgt kein Umkopieren beim Einfügen und Löschen von Elementen.

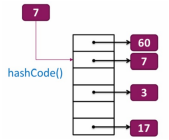
4.4.1 LinkedList: Kosten

Operation	Methode	Effizienz
Index-Zugriff	get(), set()	Langsam (traversieren)
Hinzufügen	add()	Sehr schnell (Knoten einhängen)
Entfernen	remove(int)	Langsam in Mitte Sehr schnell am Anfang und Ende
Finden	contains()	Langsam (traversieren)

4.5 HashSet vs. TreeSet

- Sets sind Container für Mengen
- Duplikate **nicht** erlaubt
- Gleichheit wird mit `equals()` geprüft.
- HashSets sind **unsortiert** und **oft sehr effizient**

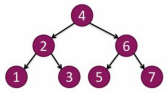
```
Set<String> firstSet = new HashSet<>();
```



Elemente liefern `hashCode()` konsistent zu `equals()`

TreeSets sind **sortiert** und **immer effizient**

```
Set<String> firstSet = new TreeSet<>();
```



Elemente impl. `Comparable` und `equals()`

4.5.1 HashSet vs. TreeSet: Kosten

Operation	TreeSet	HashSet
Finden	Schnell	Sehr schnell
Einfügen	Schnell	Sehr schnell
Löschen	Schnell	Sehr schnell (nur bei "guter" Impl.)
Sortierung	Ja	Nein

4.6 HashMap vs. TreeMap

- Maps sind für Mengen von Schlüssel-Wert-Paaren.
- Jedem Schlüssel ist genau ein Wert zugeordnet.
- **Keine** doppelten Schlüssel erlaubt.

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

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

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

Student st = map.get(20000);

for (Student s : map.values()) {
  System.out.println(s);
}
```

Schlüssel: Matrikelnummer
Wert: Student

Suchen nach Schlüssel

Collection aller Werte

HashMaps sind **unsortiert** und **oft sehr effizient**

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

TreeMaps sind **nach Schlüssel sortiert** und **immer effizient**

```
Map<Integer, Student> masters = new TreeMap<>();
```

4.6.1 HashMap vs. TreeMap: Kosten

Operation	TreeMap	HashMap
Finden	Schnell	Sehr schnell
Einfügen	Schnell	Sehr schnell
Löschen	Schnell	Sehr schnell (nur bei "guter" Impl.)
Sortierung	Ja, nach Schlüssel	Nein

4.7 equals(), Hashing

- Operator `==` liefert einen Referenzvergleich
- Methode **equals()** ist für den inhaltlichen Vergleich.
- Alle Klassen erben equals() von *Object*.
  Die Default-Implementation liefert `a == b` (Referenzvergleich).
- Bei einigen Klassen, z.B. *String, Integer, ...*
  ist equals() bereits überschrieben.
- Eine Klasse **muss** equals() von *Object* überschreiben:

```
class Person { ...

  @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 Objects.equals(firstName, other.firstName)
           && Objects.equals(lastName, other.lastName);
  }
}
```

Referenzvergleich, auf null und Klasse prüfen

Funktioniert auch bei null-Werten

4.7.1 Hashing: Konzept

- Funktion `hashCode()` bildet ein Objekt auf seinen Hash-Code ab, welcher den Ablegeort des Objekts definiert.
- Gleiche Objekte können den gleichen Hashcode haben.

Gleicher Hashcode muss aber nicht gleiches Objekt sein!

4.7.2 Hashfunktion → Für jedes equals() ein hashCode() schreiben

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

31 * firstName.hashCode() + lastName.hashCode()

5 Vererbung

- Vererbung funktioniert in Java ähnlich wie in C++
- Eine Klasse in Java kann jedoch nur **eine** Basisklasse haben
- Abgeleitete Klasse erbt Instanzvariablen und Instanzmethoden der Basisklasse
- Die oberste Klasse aller Basisklassen ist `Object`
- Methoden von `Object` sind:
  - ◇ `public String toString()`
  - ◇ `public boolean equals(Object obj)`
  - ◇ `public int hashCode()`
  - ◇ ...

5.1 Impliziter Code in Vererbung

- Hervorgehobener Code wird automatisch erstellt

```
public class Vehicle extends Object {
   private int speed;

   public Vehicle() {
     super();
     speed = 0;
   }
}

public class Car extends Vehicle {
   private boolean[] isDoorOpen;
   public Car() {
     super();
   }
}
```

Implizites Erben von Object

Impliziter Aufruf des Basis-Konstruktors, Initialisierung Instanzvariable mit Default

Impliziter Default-Konstruktor

5.2 Konstruktor bei Vererbung

```
public class Vehicle {
   private int speed;
   public Vehicle(int speed) {
     this.speed = speed;
   }
}

public class Car extends Vehicle {
   private boolean[] isDoorOpen;

   public Car(int speed, int nofDoors) {
      super(speed);
      isDoorOpen = new boolean[nofDoors];
   }
}
```

Muss erste Anweisung sein

5.3 Overriden von Methoden

Gleiche Funktion wie das Keyword `virtual` in C++, dieses gibt es jedoch nicht in Java. Stattdessen wird vor einer neu implementierten Methode einer Subklasse das Schlüsselwort `@Override` gesetzt. Dies ist optional, aber sinnvoll.

```
@Override
public void print()
```

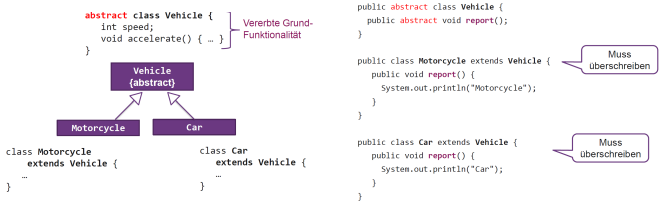
Mit `super` wird eine überschriebene Methode aufgerufen.

```
class Vehicle {
  public void report() {
    System.out.print("This is a vehicle");
  }
}

class Car extends Vehicle {
  public void report() {
    super.report();
    System.out.print("This is a car");
  }
}
```

5.4 Abstrakte Klassen → abstract

- Eine abstrakte Klasse ist nicht vollständig implementiert, sprich einzelne Methoden können nicht implementiert sein
- Die Klasse kann nicht instanziiert werden.
- Dient als Basistyp für Sub-Klassen (statischer Typ)
- Vererbt ihre Grundfunktionalität an Sub-Klassen



6 Binding

6.1 Dynamic Binding

Generell bei nicht-privaten Instanzmethoden

6.2 Static Binding

Generell bei privaten Instanzmethoden (In Subklasse nicht mehr sichtbar → Neudef. der Methode) und statischen Methoden

7 Schnittstellen

- Dient als Schleuse zwischen Klasse und Aussenwelt.
- Die Klasse muss die Funktionalität *implementieren*
- Die Aussenwelt darf die Funktionalität *nutzen*.

- Methode(n) einer Schnittstelle: implizit `public` und `abstract`.
- Deshalb werden nur **Methodendeklarationen** aufgeführt, alles andere ist ungültig/unnötig.

- Diese Taktik wird als **lose Kopplung** bezeichnet.
- Erlaubt unabhängige Entwicklung verschiedener Teams.

- Mehrere Klassen können eine Schnittstelle implementieren
- Klasse kann aber auch mehrere Schnittstellen implementieren → **Mehrfach-Implementierung** erlaubt.

7.1 Abstrakte Klassen vs. Interfaces

Abstrakte Klassen	⟷	**Interfaces**
(siehe auch 5.4) enthalten Instanzvariablen, Konstruktoren und teilweise implementierte Methoden		enthalten nur Deklarationen, keinen Code

Wann Interfaces?

- Implementierung (noch) nicht bekannt
- Implementierungen haben wenig gemeinsamen Code
- Losere Kopplung

Wann abstrakte Klassen?

- Code bei mehreren Klassen wiederverwenden
- Klassen haben gemeinsame Instanzvar. und Methoden
- Konstruktor erforderlich, um Instanzvar. zu init.

7.2 Ein Interface - mehrere Implementierung

```
interface Vehicle {
  void drive();
  int maxSpeed();}
```

```
class RegularCar implements Vehicle {
    // ...
    public void drive() {
        System.out.println("drive");
    }

    public int maxSpeed() {
        return 120;
    }
}
```

```
class RacingCar implements Vehicle {
    // ...
    public void drive() {
        System.out.println("race");
    }

    public int maxSpeed() {
        return 360;
    }
}
```

7.3 Mehrere Interfaces - eine Implementierung

```
interface Vehicle {
  void drive();
  int maxSpeed();}
```

```
interface House {
  int nOfBeds();
  void useKitchen();}
```

```
class MobileHome implements Vehicle, House {
    // ...
    public void drive() { ... }
    public int maxSpeed() { return 80; }
    public int nOfBeds() { return 3; }
    public void useKitchen() { ... }
}
```

8 Exceptions, Errors

Errors:	• Schwerwiegende Fehler, **nicht behandeln!** • Fehler in JVM: OutOfMemoryError, ... • Programmierfehler: AssertionError
Exceptions:	• Laufzeitfehler, die **behandelbar** sind • fehlerhafte Bedienung, Parameter, ... • siehe auch Checked / Unchecked 8.0.1

8.0.1 Checked, Unchecked

Checked

- Exception muss behandelt werden ODER
- `throws`-Deklaration im Methodenkopf
- Wird vom Compiler geprüft

Unchecked

- Kein `throws` oder Behandlung nötig
- `RuntimeException` und `Error`, sowie ihre Unterklassen
- Wird nicht vom Compiler geprüft

8.1 Exception auslösen

- Im Fehlerfall Exception werfen

```
String clip(String s) throws Exception {
  if (s == null) {
    throw new Exception("String is null");
  }
  if (s.length() < 2) {
    throw new Exception("String is too short");
  }
  return s.substring(1, s.length()-1);
}
```

Aufrufer soll wissen, dass es diese Exception geben kann

Exception auslösen

Fehlerangabe für menschlichen Leser

8.1.1 throws-Deklaration

- Methode muss alle potentiellen Exceptions deklarieren, die der Aufrufer erhalten könnte. (Ausser Unchecked 8.0.1)
- Aufrufer muss die Exception behandeln (fangen) / weiterreichen.

8.2 Exception behandeln

- Regulärer Code (`try`-Block)
- Fehlerbehandlung (`catch`-Block)
  - Exception im `try`-Block → `catch`-Block
  - Keine Exception: `catch`-Block wird nicht ausgeführt
- Opt. `finally`-Block: Wird immer durchlaufen

```
try {
  // …
} catch (…) {
  // …
} finally {
  // … statements
}
```

- Mehrere `catch`-Blöcke: **Nur** der erste Passende (von oben nach unten gesucht) ausgeführt.

- Falls Exception nicht behandelt wird → wird die Exception zum nächsthöheren Aufrufer geschickt.
- Wenn dies `main()` ist, wird die Exception an die JVM geworfen und das Programm abgebrochen.

8.2.1 try-with-resources → für Obj. die geschlossen werden müssen

(Interface `AutoCloseable`)

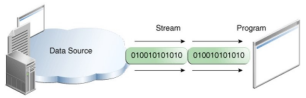
```
try (Scanner s = new Scanner(System.in)) {
  // work with s
}
```

äquivalent

```
Scanner s = new Scanner(System.in);
try {
    // work with s
} finally {
    if (s != null) { s.close(); }
}
```

9 I/O-Streams java.io.*

Input Stream	**Output Stream**
Daten **von aussen** lesen • Tastatur • Netzwerk • Dateien	Daten **nach aussen** schreiben • Bildschirm/Konsole • Netzwerk • Dateien



Byte-Streams	**Character-Streams**
• 8-Bit-Daten • Klassen erben von `InputStream, OutputStream`	• 16-Bit Textzeichen (UTF-16) • Klassen erben von `Reader, Writer` • Zeichen- / Zeilenweise Ein- & Ausgabe

9.1 Byte-Streams

InputStream: `int read(byte[] b, int offset, int length)`
OutputStream: `void write(byte[] b, int offset, int length)`
Lese/schreibe `length` Bytes in Array `b` ab Index `offset`

`void flush()`: Implizit bei `close()`

9.1.1 Standard Input/Output

`System.in` → *InputStream*

`System.out`, `System.err` → *PrintStream* (Subklasse von *OutputStream*)

9.1.2 FileInput: Ganze Datei binär einlesen (kann speicherintensiv werden)

```
byte[] data = Files.readAllBytes(Path.of("in.bin"));
```

```
var in = new FileInputStream("myFile.data");
int value = in.read();
while (value >= 0) {
  byte b = (byte)value;
  // work with b
  value = in.read();
}
in.close();
```

Bestehende Datei zum Lesen öffnen

-1: end of file

Gelesenes Byte

9.1.3 FileOutput: Ganze Datei binär schreiben

```
Files.write(Path.of("out.bin"), data);
```

```
var out = new FileOutputStream("test.data");
while (…) {
  byte b = …;
  out.write(b);
}
out.close();
```

Datei neu anlegen bzw. überschreiben

Schreiben («Flush») des Rests beim Schliessen

`new FileOutputStream("test.data", true)` um an existierende Datei anzuhängen

9.2 Character-Stream

9.2.1 FileReader

```
try (var reader = new FileReader("quotes.txt", StandardCharsets.UTF_8)) {
  int value = reader.read();
  while (value >= 0) {
    char c = (char)value;
    // use character
    value = reader.read();
  }
}
```

-1 = end of file

UTF-8 Codierung

16-bit char

```
new FileReader(f)
↕
new InputStreamReader(new FileInputStream(f))
```

9.2.2 FileWriter

```
try (var writer = new FileWriter("test.txt",
                  StandardCharsets.UTF_8, true)) {
  writer.write("Hello!");
  writer.write('\n');
}
```

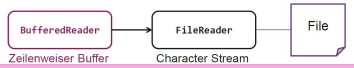
Append

String schreiben

Einzelnen char schreiben

9.2.3 Zeilenweises Lesen

```
try (var reader = new BufferedReader(new FileReader("quotes.txt")) {
  String line = reader.readLine();
  while (line != null) {
    System.out.println(line);
    line = reader.readLine();
  }
}
```



9.2.4 Einfachster Text-Datei-Zugriff

Ganze Text-Datei lesen

```
List<String> lines = Files.readAllLines(Path.of("in.txt"),
StandardCharsets.UTF_8);
```

Ganze Text-Datei schreiben

```
Files.write(Path.of("out.txt"),
lines, StandardCharsets.UTF_8);
```

9.3 Serialisierung

`Serializable`-Interface implentieren
`class Person implements Serializable {..}`
→ Klassen in Files einfach abzuspeichern & wieder reinzuladen
ACHTUNG: Wird die Klasse vor dem Deserialisieren abgeändert, z.B. eine neue Variable, funktioniert dies nicht!

```
var person = new Person();

// Objekt serialisieren
try(var stream = new ObjectOutputStream(
                 new FileOutputStream("serial.bin"))) {
    stream.writeObject(person);}

// Objekt deserialisieren
try(var stream = new ObjectInputStream(
                 new FileInputStream("serial.bin"))) {
    Person p = (Person)stream.readObject();}
```

10 Threads

Schnellere Programme:

- Aufteilen einer Datenmenge in mehrere Teile, um parallel zu arbeiten
- Teilresultate nach Verarbeitung zusammenführen
- Bsp.: Sortierverfahren, Komprimierung, Bildverarbeitung

Einfachere Programme:

- Gleichzeitig oder verzahnt ausführbare Abläufe
- Bsp.: Layout, Speichern im Hintergrund

Multi-Core-Prozessoren können mehrere Threads parallel ausführen, für jeden Core zwei Threads.

10.1 JVM Thread Modell

Java ist ein Single Process System. JVM erzeugt beim Aufstarten einen Thread, welcher `main()` aufruft. Der Programmierer, Subsysteme oder das Laufzeitsystem können ebenfalls Threads starten.
Die JVM läuft, solange Threads laufen, ausser wenn Threads als *Daemon* markiert sind (z.B. Garabge Collector). JVM wartet nicht auf Daemon Threads, diese werden bei JVM-Ende unkontrollliert abgebrochen.

10.1.1 Runnable Interface

```
@FunctionalInterface
interface Runnable {
   public void run();
}
```

Kann auch als Lambda übergeben werden:

```
Thread(Runnable task)
```

Funktionsschnittstelle eines Threads, sie wird beim Starten des Threads durch die JVM gerufen.

10.1.2 Start und Ende

Ein Thread wird nach `start()` ausgeführt (über `run()` des `Runnable`-Interfaces):

```
var myThread = new Thread(()-> {
    // Implementation
});

myThread.start(); // Start nach Erzeugung
```

Der Thread endet beim Verlassen von `run`, z.B. durch Ende der Methode, Return Statemnt oder unbeh. Exception

10.2 Multi-Thread Beispiel

```
public class Demo02MultiThread {
    public static void main(String[] args) {
        var thread1 = new Thread(() -> print("A"));
        var thread2 = new Thread(() -> print("B"));
        thread1.start();
        thread2.start();
        System.out.println("main finished");
    }
        static void print(String label) {
            for (int i = 0; i<5; i++) {
                System.out.println(label + ": " + i + ",");
        }
    }
}
// Ausgabe: undefiniert
```

10.3 Alternative Implementationen

10.3.1 Explizit

```
class MyRunnable implements Runnable {
    @Overridepublic void run(){
        // Thread code
    }
}
var myThread = new Thread(new MyRunnable());
myThread.start();
```

10.3.2 Sub-Klasse von Thread

```
class SimpleThread extends Thread{
    @Override
    public void run(){
        // Thread code
    }
}
var myThread = new SimpleThread();
myThread.start();
```

10.4 Thread-Methoden

10.4.1 Thread Passivierung

Thread.sleep(milliseconds) Laufender Thread wird schlafen gelegt

Thread.yield() Laufender Thread gibt Prozessor frei und wird wieder ablaufbereit

10.4.2 InterruptedException

Mögliche Exception bei blockierenden Aufrufen, z.B. `join()`, `sleep()`, ...
Threads können von aussen unterbrochen werden: `myThread.interrupt()` → bricht `join()`, `sleep()`, ... ab

10.4.3 Weitere Methoden

`static Thread currentThread()` liefert Instanz des Threads

`long threadId()` liefert ID des Threads

`void setDaemon(boolean on)` Thread als *Daemon* markieren

10.5 Synchronisation

Threads teilen sich Adressraum und Heap. Wird auf dasselbe Objekt zugegriffen, muss abgesichert werden.
`synchronized` ist ein Modifier für Methoden, ähnlich wie ein Flag.

```
class BankAccount {
    private int balance = 0;
    public synchronized void deposit(int amount) {
        balance += amount;
    }
    public synchronized boolean withdraw(int amount) {
        if (amount <= balance) {
            balance -= amount;
            return true;
        } else {
            return false;
        }
    }
}
```

Nur ein Thread kann eine der `synchronized`-Methoden zur gleichen Zeit in derselben Instanz ausführen

11 Stream-API

`java.util.stream.*`
Wird für deklarative Abfragen von Collections gebraucht. Code definiert, **was** gesucht wird, nicht **wie**. Das Framework arbeitet sehr intensiv mit Lambdas und ist komplett unabhängig von Input/Output-Streams.
Basisschnittstellen:

- Für Referenzdatentypen

  - `Stream<T>`
- Für primitive Datentypen
  - `IntStream`
  - `LongStream`
  - `DoubleStream`

11.1 Endliche Quellen

`list.stream()`	Liefert Stream anh. Collection
`Arrays.stream(array)`	Liefert Stream anh. Array
`IntStream.range(1, 100)`	Zahlen von 1 bis 100
`Stream.of(2, 3, 5, 7)`	eigene Aufzählung
`Stream.concat(strm1, strm2)`	Verketteter Stream

11.2 Unendliche Quellen

`generate()`

```
Random random = new Random();
Stream.generate(random::nextInt)
  .forEach(System.out::println);
```

`iterate()`

```
IntStream.iterate(0, i -> i + 1)
  .forEach(System.out::println);
```

11.3 Zwischenoperationen

`filter(Predicate)`	Filtern mit Lambda
`map(Function)`	Mappen mit Lambda
`mapToInt(Function)`	Mappen mit `int,long, double`
`sorted(Comparator)`	Sortieren mit Comparator
`distinct()`	Duplikate entfernen gemäss `equals()`
`limit(long n)`	n-Elemente liefern
`skip(long n)`	n-Elemente überspringen

Zwischenoperationen dürfen die Collection **nicht ändern** und sie dürfen keine Abhängigkeit zu äusseren, änderbaren Variablen haben.

11.4 Terminaloperationen

Der Stream ist nach einem solchen Aufruf fertig

`forEach(Consumer)`	Pro Element Operation anwenden
`count()`	Anzahl Elemente
`min(),max()`	bei `Stream<T>` Comparator-Arg. erf.
`average(), sum()`	Nur bei in/long/double-Stream
`findAny(), findFirst()`	Gibt irgendein/erstes Element zurück
`collect()`	Rückumwandlung zu Collection
`toArray()`	Rückumwandlung zu Array

11.4.1 Collectors

`Collectors.toList()` → in Liste abbilden
`Collectors.toCollection(TreeSet::new)`
↪ in beliebige Collection abbilden (Konstruktorreferenz)
`Collectors.groupingBy(key, aggregator)`:

- Gruppierung mit opt. Aggregator
- Aggregator: averaging, summing, counting
- Liefert HashMap als Rückgabewert

11.5 Funktionsschnittstellen

11.5.1 Vordefinierte

```
filter(Predicate<T> p)
```

```
interface Predicate<T> {
  boolean test(T input);
}
```

```
map(Function<T, R> f)
```

```
interface Function<T, R> {
  R apply(T input);
}
```

```
forEach(Consumer<T> c)
```

```
interface Consumer<T> {
  void accept(T input);
}
```

11.5.2 Optional-Wrapper

`average(), min(), max(), findAny(), findFirst()`

```
OptionalDouble averageAge =
    people.stream().mapToInt(p -> p.getAge()).average();

if (averageAge.isPresent()) {
    double result = averageAge.getAsDouble();
    System.out.println(result);
}
```

11.5.3 Matching

`allMatch(), anyMatch(), noneMatch()`
Prüfen, ob das Prädikat auf alle / irgendein/ kein Element zutrifft

```
boolean 18plus = ppl.stream().allMatch(p->p.getAge >= 18);
```

12 Code-Snippets

12.1 Konsolentext

```
public static void main(String[] args) {
    try (var scanner = new Scanner(System.in)) {
        System.out.println("Text eingeben: ");
        String inputText = scanner.next();
        System.out.println("Neuer Text: ");
        String inputText2 = scanner.next();
        String both = inputText + inputText2;
        System.out.println(both);
    }
}
```

12.2 Fakultät

```
public class Factorial {
    public static void main(String[] args) {
        int n = 12;
        int p = 1;
        int i = 1;
        while (i <= n) {
            p = p * i;
            ++i;
        }
        System.out.println(p);
    }
}
```

12.3 Array-Loop

```
int[] array = {1, 2, 3};

for(int i=0; i < array.length; i++){
    System.out.printf("Index %d : %d\n", i, array[i]);
}

// Index 0 : 1
// Index 1 : 2
// Index 2 : 3

int[][] m = new int[2][3];

// int[Zeile][Spalte]
// m.length    => Anzahl Zeilen
// m[0].length => Anzahl Spalten
```

12.4 Schleife mit continue

```
// Zahlen aufsummieren, die groesser 0 sind
int[] numbers = { -5, 4, 2, -7, 4 };
int sum = 0;

for (int x : numbers) {
    if (x < 0) {
        System.out.println("Verworfen: " + x);
        continue;
    }
    sum += x;
}
System.out.println("Summe: " + sum);
```

12.5 Unit Test

```
public class Demo07AbsTest {
    @Test
    @DisplayName("Positive Number")
    void testPositiveValue() {
        long in = 23;
        long out = 23;
        assertEquals(out, Demo07Abs.abs(in));
    }

    @Test //Exception muss auftreten => PASS
    @DisplayName("Max Negative Number")
    void testMaxNegativeValue() {
        long in = Long.MIN_VALUE;
        assertThrows(RuntimeException.class, () -> Demo07Abs.abs(
     in));
    }
}
```

12.6 Getter- und Settermethoden

```
private int width; // Geschuetzt

// Getter
public int getWidth(){ return width; }

// Setter, Wert zuerst pruefen
public void setWidth(int width){
    if(width < 0){
        throw new IllegalArgumentException();
    }
    this.width = width;
}
```

12.7 Concatenate Strings

```
public class VariadicMethodString {

  public static String concat(String delim, String ...texts) {
    StringBuilder buf = new StringBuilder();
    for (int i=0; i<texts.length; i++) {
      buf.append(texts[i]);
      if( i < texts.length -1 ) {
        buf.append(delim);
      }
    }
    return buf.toString();
  }

  public static void main(String[] args) {

    String cats = concat(", ", "Bella", "Dana", "Tom");
    System.out.println(cats); // Output: Bella, Dana, Tom

    String people = concat(";", "Hans", "Claudia", "Felix", "Maria
     ");
    System.out.println(people); // Output: Hans;Claudia;Felix;
     Maria

  }

}
```

12.8 Exceptions

```
public static void foo() {
    try{
        System.out.println("A");
        throw new Exception("B");
        System.out.println("C");
    } catch (Exception e) {
        System.out.println("Catch");
        System.out.println(e);
    } finally {
        System.out.println("Finally");
    }
} // Ausgabe: A Catch java.lang.Exception:B Finally
}
```

12.9 Methodenreferenz, Lambda

```
@FunctionalInterface // Funktionsschnittstelle
public interface Predicate<T>{
    boolean test(T element);
}

public class Utils { // Implementation
    static <T> void removeAll(Collection<T> collection, Predicate<
     T> criterion){
        var it = collection.iterator();
        while(it.hasNext()){
            if(criterion.test(it.next())){
                it.remove();
            }
        }
    }
}

// Als Lambda
Utils.removeAll(people, p->p.getAge() > 65);

// Als Methodenreferenz
Utils.removeAll(people, this::isSenior);

boolean isSenior(Person person){
    return person.getAge() > 65;
}
```

12.10 Randomizer

```
// Muenzwurf-Programm
var random = new Random();
long amount = Stream
    .generate(() -> random.nextBoolean())
    .limit(1000)
    .filter(x -> x)
    .count();
System.out.println(amount);
```