Java für C++-Programmierer

HS 2024, Peter Bühler

Fabian Suter, 8. Dezember 2024

0.0.1

1 Basics

1.1 Grundsätzlich

Diese Zusammenfassung dient zur Hilfe beim Programmieren mit Java anhand von C++-Vorwissen. Es gibt einige wichtige Unterschiede, **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 Sourcecode Hello world Java

```
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

1.3 Java Runtime

Der Compiler erzeugt anders als in C++ **Bytecode**, welcher anschliessend auf einer Java Virtual MAchine (JVM) laufen kann. Dadurch wird Java plattformunabhängig, da jede Plattform eine JVM hat, sei es Windows, Android oder MacOS.

2 Datentypen

Ein Datentyp besteht aus *Werten* und *Operationen*. Java besitzt zwei generelle Datentypen, namentlich **Primitive Datentypen** und **Referenzdatentypen**.

2.1 Primitive Datentypen

Sie haben plattformunabhängig den gleichen Wertebereich, es gibt keine `unsigned`-Typen. Für bool'sche Werte gibt es den Datentyp `boolean`.

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

Bei **Ganzzahlen** ist der Überlauf in Java definiert, im Gegensatz zu C++.
2147483647 + 1 → -2147483648

Bei **Gleitkommazahlen** gilt dasselbe:
2 * 1e308 → `POSITIVE_INFINITY`
5e-324 / 2 → 0.0

2.1.2 Undefinierte Operationen

Ganzzahlen werfen bei Division/Modulo durch 0 einen Fehler bzw. eine Exception.

Gleitkommazahlen werfen bei Division durch 0 ein `POSITIVE_INFINITY` resp. `NEGATIVE_INFINITY`
Bei undefinierten Rechnungen wie 0 / 0 wird `NaN` zurückgegeben.

2.1.3 Text-Literale

char mit Apostrophen: `'A'`, `'\n'` (NewLine), `'\''` (Apostroph)

String mit Anführungszeichen: `"Say \"hello\"!\n"`

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 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 **9.2**.

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

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

Siehe auch Kapitel 4

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
```

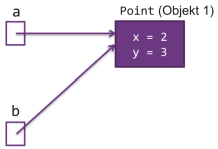
3 Objektorientierung

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:

```java
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

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

// x bleibt 5
```

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

Referenzdatentypen:
Methode arbeitet mit Kopie der Referenz

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

// a geändert
```

```java
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*

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

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

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

3.3.1 Lambdas

Ad-Hoc Implementierung anstatt einer expliziten Methodenreferenz.

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

`Person`,`{}` und `return` sind optional, wenn nur ein Ausdruck enthalten ist
Faustregel: Lambdas sind kurz (∼ 3 Zeilen), für längere Methodenreferenzen verwenden, siehe auch 9.8

3.4 Unit Testing

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) und Edge Cases (z.B. 0, max. und min. Bereich, usw.). Siehe auch 9.4
Faustregel:

- 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

```java
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 9.5

3.5 Generics

Disclaimer, folgende Themen sind hier nicht enthalten: Typebounds, Wildcardtypen, Covarianz, Type-Erasure
Generics dienen dazu, typ-unabhängige Frames zu erstellen.

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

Eine Klasse mit Typ-Variable. Die Variable dient als Platzhalter für unbekannten Typ. Verschachtelung bei der Anwendung ist zulässig.

```java
class Stack<T> {
  // …
}
```
Typ-Variable

```java
Stack<String>  stack1;
Stack<Integer> stack2;
```
Typ-Argument

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

Generische Methoden sind unabhängig von generischen Klassen und Interfaces. Die 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) { ... }
```

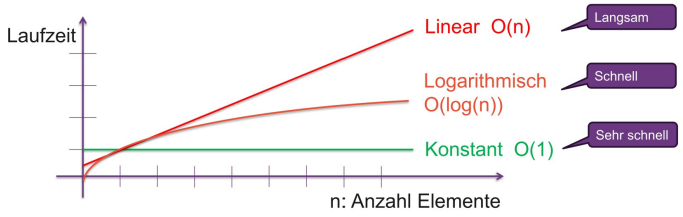
4 Collections

In Collections können nur Referenzdatentypen abgelegt werden. Beim Hinzufügen des Elements wird das Objekt selber **nicht** kopiert, es wird nur eine Referenz 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

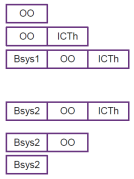
```
// Boxing
int x = 123;
Integer obj = Integer.valueOf(x);

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

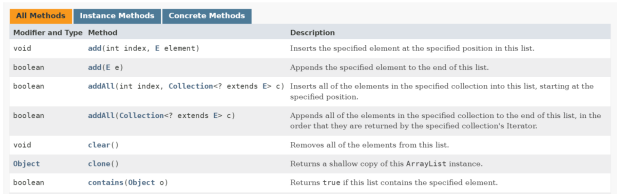
Primitiver Typ	Wrapper-Klasse
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

4.3 ArrayList

ArrayLists sind eine geordnete Folge von Elementen mit demselben Referenzdatentyp. Elemente können einfach hinzugefügt oder entfernt werden. Duplikate oder `null`-Einträge sind möglich
Der 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 und 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
```





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.

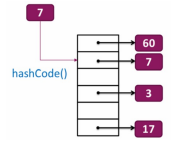
4.4.1 LinkedList: Kosten

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, wobei Duplikate **nicht** erlaubt sind. Die Gleichheit wird mit `equals()` geprüft.
HashSets sind **unsortiert** und **oft sehr effizient**

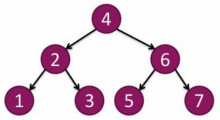
```
Set<String> firstSet = new HashSet<>();
```



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

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

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



Elemente implementieren `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. Es sind **keine** doppelten Schlüssel erlaubt.

Beispiel:

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

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

map.put(20000, st1);                    } Schlüssel: Matrikelnummer
map.put(70000, st2);                    } Wert: Student

Student st = map.get(20000);            } Suchen nach Schlüssel

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

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

Der Operator `==` liefert einen Referenzvergleich, die 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

Die Funktion `hashCode()` bildet ein Objekt auf seinen Hash-Code ab, welcher den Ablegeort des Objekts definiert.
Gleiche Objekte können den gleichen Hashcode haben.
ACHTUNG : gleicher Hashcode muss aber nicht gleiches Objekt sein!

4.7.2 Hashfunktion

Als Faustregel: Bei jedem **equals()** gleich ein **hashCode()** schreiben:

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

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

5 Vererbung

Die Vererbung funktioniert in Java ähnlich wie in C++. Eine Klasse in Java kann jedoch nur **eine** Basisklasse haben. Eine abgeleitete Klasse erbt die 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()
```

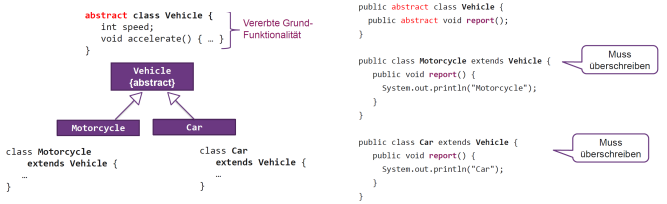
Mit `super` wir eine überschriebene Methode aufgerufen.

```
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

Schlüsselwort: `abstract`

Eine abstrakte Klasse ist nicht vollständig implementiert, sprich einzelne Methoden können nicht implementiert sein und die Klasse kann nicht instanziiert werden.
Sie dient als Basistyp für Sub-Klassen (statischer Typ) und vererbt ihre Grundfunktionalität an Sub-Klassen.
Beispiel:



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

Eine Schnittstelle dient als Schleuse zwischen Klasse und Aussenwelt. Die Klasse muss die Funktionalität *implementieren*, die Aussenwelt darf die Funktionalität *nutzen*.
Die Methode(n) einer Schnittstelle sind implizit `public` und `abstract`. Deshalb werden nur **Methodendeklarationen** aufgeführt, alles andere ist ungültig/unnötig.

Diese Taktik wird als **lose Kopplung** bezeichnet und erlaubt unabhängige Entwicklung verschiedener Teams.

Mehrere Klassen können eine Schnittstelle implementieren, eine Klasse kann aber auch mehrere Schnittstelle 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("driving");
  }

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

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

  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

Die Methode muss alle potentiellen Exceptions deklarieren, die der Aufrufer erhalten könnte. (Ausser Unchecked 8.0.1)
Der Aufrufer muss die Exception behandeln (fangen) oder 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
}
```

Bei mehreren `catch`-Blöcken wird **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 Objekte, 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 Code-Snippets

9.1 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);
    }
}
```

9.2 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
```

9.3 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);
```

9.4 Unit Test

```
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));
    }

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

9.5 Getter- und Settermethoden

```
private int width; // Geschuetzt

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

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

9.6 Concatenate Strings

```
package javac.v4;

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

  }

}
```

9.7 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
```

9.8 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;
}
```