

# FOP Reference Sheet

Jonas Milkovits

Last Edited: 27. April 2020

## Inhaltsverzeichnis

1 Collections	1
2 Computerspeicher	4
3 Datenstrukturen	4
4 Datentypen	5
5 Exceptions (java.lang.Exception;)	6
6 Fehler	7
7 Files	7
8 Functional Interfaces und Lambda-Ausdrücke	11
9 Graphical User Interface	11
10 Generics	23
11 Graphics (java.awt.Graphics;)	25
12 Interfaces	25
13 JUnit-Tests	26
14 Klassen	26
15 Konversionen	27
16 Methoden	28
17 Optional (java.lang.Optional;)	29
18 Packages und Zugriffsrechte	29
19 Programme und Prozesse	30
20 Random (java.util.Random;)	30
21 Schleifen, if, switch	30
22 Streams (java.util.stream.Stream;)	31
23 String (java.lang.String)	32
24 Syntax	32

<b>25 Threads</b>	<b>33</b>
<b>26 Vererbung</b>	<b>35</b>
<b>27 Anhang: Interne Zahlendarstellung</b>	<b>36</b>
<b>28 Anhang: Korrekte Software</b>	<b>38</b>
<b>29 Anhang: Effizienz von Software</b>	<b>41</b>
<b>30 Anhang: Fehlersuche und fehlervermeidender Entwurf</b>	<b>47</b>
<b>31 Anhang: Polymorphie</b>	<b>51</b>

# 1 Collections

Informationen	<ul style="list-style-type: none"><li>▷ Sammlungen von Elementen (Objekte eines generischen Typs)</li><li>▷ Struktur:<ul style="list-style-type: none"><li>◊ Alle Klassen und Interfaces in <code>java.util</code></li><li>◊ Interface <code>Collection</code>: Alle Klassen implementieren dieses Interface</li><li>◊ Klasse <code>Collections</code>: Basisalgorithmen, Sortieren</li><li>◊ Interface <code>List</code>: Erweitert <code>Collection</code>, mehr Funktionalitäten</li><li>◊ Klasse <code>Iterator</code>: Iteration über die Elemente einer <code>Collection</code></li></ul></li><li>▷ Beispiele für Klasse, die das Interface <code>Collection</code> implementieren:<ul style="list-style-type: none"><li>◊ <code>Vector</code>, <code>LinkedList</code>, <code>ArrayList</code>, <code>TreeSet</code>, <code>HashSet</code></li></ul></li></ul>
Interface <code>Collection</code>	<ul style="list-style-type: none"><li>▷ z.B.: <code>Collection&lt;Number&gt; c1 = new ArrayList&lt;Number&gt;();</code><ul style="list-style-type: none"><li>◊ Speichert leere <code>ArrayList</code> in einer Referenz des Interface <code>Collection</code></li><li>◊ Dies ist möglich, da <code>ArrayList</code> das Interface <code>Collection</code> implementiert</li></ul></li><li>▷ Methoden:<ul style="list-style-type: none"><li>◊ <code>add</code><ul style="list-style-type: none"><li>- Fügt zur <code>ArrayList</code> ein neues Element hinzu</li><li>- Gibt <code>true</code> zurück, falls Hinzufügen erfolgreich</li></ul></li><li>◊ <code>addAll</code><ul style="list-style-type: none"><li>- Hat eine <code>Collection</code> als Parameter und fügt diese hinzu</li></ul></li><li>◊ <code>size</code><ul style="list-style-type: none"><li>- Anzahl der Elemente als <code>int</code></li></ul></li><li>◊ <code>isEmpty</code><ul style="list-style-type: none"><li>- <code>true</code>, falls <code>Collection</code> keine Elemente enthält (<code>size == 0</code>)</li></ul></li><li>◊ <code>contains</code><ul style="list-style-type: none"><li>- Parameter vom Typ <code>Object</code></li><li>- Überprüft, ob aktueller Parameter in <code>Collection</code> vorhanden ist</li><li>- Nutzt <code>equals</code> von <code>Object</code> → Wertgleichheit</li></ul></li><li>◊ <code>containsAll</code><ul style="list-style-type: none"><li>- <code>true</code>, falls ganze übergebene <code>Collection</code> enthalten ist</li></ul></li><li>◊ <code>clear</code><ul style="list-style-type: none"><li>- Entfernt alle Elemente aus der <code>Collection</code></li></ul></li><li>◊ <code>remove</code><ul style="list-style-type: none"><li>- Entfernt übergebenes <code>Object</code></li><li>- <code>true</code>, falls <code>Object</code> mindestens einmal vorhanden</li><li>- Bei mehreren, entscheidet die <code>Collection</code>-Klasse welches entfernt wird</li></ul></li></ul></li></ul>
Interface <code>List</code>	<ul style="list-style-type: none"><li>▷ Erweitert das Interface <code>Collection</code></li><li>▷ Unterschied: Definition einer Reihenfolge auf den Elementen</li><li>▷ Methoden:<ul style="list-style-type: none"><li>◊ <code>indexOf</code><ul style="list-style-type: none"><li>- Liefert ersten Index zurück, an dem <code>Object</code> zu finden ist</li><li>- Liefert -1 zurück, falls Parameter nicht in Liste gefunden wird</li></ul></li><li>◊ <code>set</code><ul style="list-style-type: none"><li>- <code>T set(int index, T element) ...</code></li><li>- Ersetzt Element an Stelle <code>index</code> durch <code>element</code></li><li>- Gibt ersetztes Element zurück</li></ul></li><li>◊ <code>add</code><ul style="list-style-type: none"><li>- Identisch zu Methode <code>set</code>, jedoch ein Unterschied:</li><li>- Überschreibt das Element <b>nicht</b>, sondern fügt es vor dem Element ein</li></ul></li></ul></li></ul>
Sortieren mit <code>Comparator</code>	<ul style="list-style-type: none"><li>▷ Klasse <code>Collections</code> hat Klassenmethode <code>sort</code></li><li>▷ <code>Collections.sort(list, new MyComparator());</code><ul style="list-style-type: none"><li>◊ Erster Parameter: Zu sortierende Liste (z.B.: <code>List&lt;Student&gt; list = ...</code>)</li><li>◊ Zweiter Parameter: Selbst erstellte Sortierlogik</li><li>◊ Typparameter von <code>Comparator</code> und <code>List</code> müssen gleich sein</li></ul></li></ul>

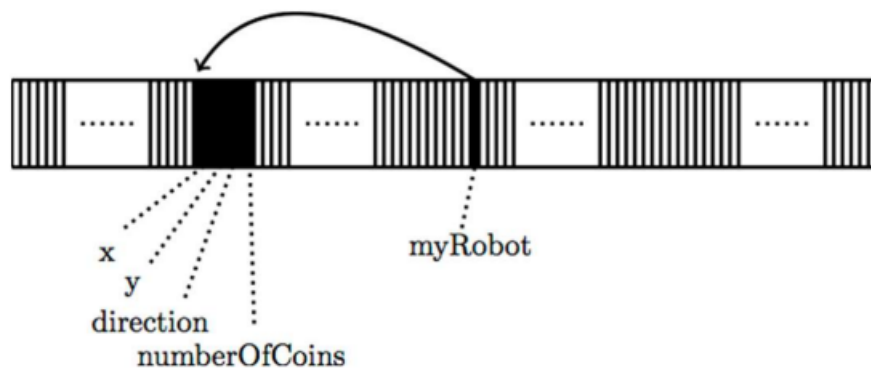
Interface <b>Iterator</b>	<ul style="list-style-type: none"> <li>▷ <b>Collection</b> und <b>List</b> erben von Interface <b>Iterable</b></li> <li>▷ Jede Klasse, die <b>Collection</b> implementiert hat eine eigene <b>Iterator</b>-Klasse</li> <li>▷ Diese eigene <b>Iterator</b>-Klasse implementiert das Interface <b>Iterator</b></li> <li>▷ <code>Collection&lt;Number&gt; c1 = new ArrayList&lt;Number&gt;();</code></li> <li>▷ <code>Iterator&lt;Number&gt; it1 = c1.iterator();</code> <ul style="list-style-type: none"> <li>◊ <b>Collection</b> besitzt die Methode <code>iterator()</code></li> <li>◊ Liefert ein Objekt ihrer eigenen <b>Iterator</b>-Klasse zurück</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>next()</code> <ul style="list-style-type: none"> <li>- Liefert ein noch nicht geliefertes Element der <b>Collection</b></li> <li>- Reihenfolge von Interface abhängig (<b>Collection</b> oder <b>List</b>)</li> </ul> </li> <li>◊ <code>hasNext()</code> <ul style="list-style-type: none"> <li>- <code>true</code>, falls mindestens ein Element noch nicht durch diesen <b>Iterator</b> zurückgeliefert wurde</li> </ul> </li> </ul> </li> </ul>
Interface <b>Map</b>	<ul style="list-style-type: none"> <li>▷ z.B.: <code>Map&lt;String,Integer&gt; map = new HashMap&lt;String,Integer&gt;();</code> <ul style="list-style-type: none"> <li>◊ Erster Typparameter: <b>Key</b> (hier: <b>String</b>)</li> <li>◊ Typparameter: <b>Value</b> (hier: <b>Integer</b>)</li> </ul> </li> <li>▷ Eine <b>Map</b> realisiert eine Abbildung von den <b>Keys</b> in die <b>Values</b> <ul style="list-style-type: none"> <li>◊ <b>Keys</b> müssen alle unterschiedlich sein</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>put(key, value)</code> // Fügt Paar in Map ein</li> <li>◊ <code>get(key)</code> // Gibt value zu bestimmtem key zurück</li> </ul> </li> </ul>

## LinkedList

- ▷ Aufbau:
  - ◊ Elemente der Liste enthalten:
    - **Key** vom Typ **T**
    - Attribut vom selben Elementtyp mit Namen **next**
  - ◊ Abspeichern des sogenannten **head**, dieser speichert die Liste
  - ◊ Die Liste wird durch die Verkettung untereinander mit **next** erstellt
- ▷ Die folgenden Beispiele sollen nur die Logik hinter der Klasse erläutern
- ▷ Durchlauf durch alle Elemente: (**LOGIK**)
  - ◊ (Die eigentliche Implementation in Java sieht anders aus)
  - ◊ `for (ListItem<T> p = head; p != null; p = p.next) {...}`
  - ◊ Setzen von **p** zu **p.next** bis **p == null**
- ▷ Einfügen Element am Anfang: (**LOGIK**)
  - ◊ Erstellen eines neuen Listitems und Kopieren der Werte
  - ◊ Achtung: Erst **head** als **next** abspeichern
  - ◊ Danach neues Listitem als **head** setzen
  - ◊ (sonst geht die komplette Liste verloren)
- ▷ Einfügen Element an Stelle n: (**LOGIK**)
  - ◊ Fortschreiten des Durchlaufs bis zu n-1
  - ◊ `ListItem<T> tmp = new ListItem<T>();`
  - ◊ `tmp.key = key; // Setzen des Keys`
  - ◊ `tmp.next = p.next; // Knüpfen des neuen Elements an n+1.Element`
  - ◊ `p.next = tmp; // Knüpfen des n-1.Elements an neues Element`
- ▷ Entfernen Element: (**LOGIK**)
  - ◊ Überspringen des zu löschenden Elements
  - ◊ `head: head = head.next;`
  - ◊ Sonst: `p.next = p.next.next;`
    - Laufpointer muss in diesem Fall eine Stelle davor stehenbleiben
- ▷ Allgemein:
  - ◊ Auf korrektes Zwischenspeichern achten!
- ▷ Doppelte Verkettung:
  - ◊ Ermöglicht rückwärts und vorwärts Durchlaufen
  - ◊ Kostet Laufzeit und Speicher
  - ◊ Verweisnamen meist **next** und **backward**
  - ◊ Erhöhter Aufwand, da doppelte Verweiskopien
- ▷ Zyklische Listen:
  - ◊ Letzter Verweis nicht **null** sondern auf **head**

## 2 Computerspeicher

Unsere Vorstellung	▷ großes Feld aus Maschinenwörtern mit eindeutiger Adresse
Erzeugung eines neuen Objekts	▷ Reservierung von ungenutztem Speicher in ausreichender Größe
Referenz	▷ Name der Variable, die die Anfangsadresse des Objekts speichert ▷ Kann auch an komplett anderer Stelle als das Objekt gespeichert sein
Speicherort primitiver Datentypen	▷ Name verweist tatsächlich auf Speicherstelle, an der Wert abgespeichert wird
Prozessablauf	▷ Program Counter enthält Adresse der nächsten Anweisung ◊ Zählt nach jeder Anwendung hoch und verweist auf nächsten Speicher ▷ CPU verarbeitet parallel die momentane Anweisung aus Program Counter
Methodenausführung	▷ Einrichtung einer Variable <b>StackPointer</b> bei Programmstart ▷ StackPointer enthält die Adresse des <b>Call-Stacks</b> ▷ Bei Methodenaufruf wird im Speicher Platz reserviert, genannt <b>Frame</b> ▷ <b>Frame</b> wird dann auf dem Call-Stack abgelegt ▷ Der <b>StackPointer</b> wird dann mit der Adresse des neuen <b>Frames</b> überschrieben ▷ Methodenaufruf vorbei: Frame wird wieder vom <b>Call-Stack</b> genommen ▷ <b>StackPointer</b> wird auf Adresse des vorherigen <b>Frames</b> gesetzt
Methodentabelle	▷ Enthält bei Objekt die Anfangsadressen der verfügbaren Methoden



## 3 Datenstrukturen

Array	▷ Verwendet zum Speichern von mehreren Variablen des selben Typs ▷ Erzeugung: <code>int[] test = new int[n];</code> ▷ <code>n</code> gibt in diesem Fall die feste Anzahl der speicherbaren Variablen an ▷ Natürlich auch Arrays von Objekten möglich ▷ Zugriff auf Variablen: <code>test[0]</code> für ersten Wert (Index) ▷ Zugriff auf Länge: <code>test.length</code>
-------	--

## 4 Datentypen

Konstanten	<ul style="list-style-type: none"><li>▷ Variable/Referenz wird dadurch unveränderbar</li><li>▷ z.B.: <code>final myClass ABC = new myClass();</code><ul style="list-style-type: none"><li>◊ Referenz zwar nicht veränderbar, Objekt aber schon</li></ul></li><li>▷ <code>Integer.MAX_VALUE</code> / <code>Integer.MIN_VALUE</code></li><li>▷ Unendlich: <code>Double.POSITIVE_INFINITY</code> / <code>Double.NEGATIVE_INFINITY</code></li><li>▷ Müssen initialisiert werden</li></ul>
Primitive Datentypen	<ul style="list-style-type: none"><li>▷ Ganze Zahlen: <code>byte</code> → <code>short</code> → <code>int</code> → <code>long</code></li><li>▷ Gebrochene Zahlen: <code>float</code> → <code>double</code></li><li>▷ Logik: <code>boolean</code></li><li>▷ Zeichen: <code>char</code></li><li>▷ Mehrere Definitionen: <code>int m = 1, n, k = 2;</code></li><li>▷ Ohne Initialisierung: undefinierter Wert</li></ul>
Literale	<ul style="list-style-type: none"><li>▷ wörtlich hingeschriebene Werte eines Datentyps</li><li>▷ Zahlen standardmäßig <code>int</code>, falls <code>long</code> gewünscht: <code>123L</code> oder <code>123l</code></li><li>▷ Bei gebrochenen <code>double</code>, falls <code>float</code> gewünscht: <code>12.3F</code> oder <code>12.3f</code></li><li>▷ <code>null</code>: Nutzung für Referenzen → verweist auf nichts</li></ul>
Boolean	<ul style="list-style-type: none"><li>▷ nur <code>true</code> und <code>false</code></li><li>▷ Negation <code>!a</code></li><li>▷ Logisches Und: <code>a &amp;&amp; b</code></li><li>▷ Logisches Oder: <code>a    b</code> (inklusive)</li><li>▷ Gleichheit: <code>a == b</code></li></ul>
Zeichentyp <code>char</code>	<ul style="list-style-type: none"><li>▷ z.B.: <code>char c = 'a';</code></li><li>▷ Interne Kodierung als Unicode</li><li>▷ <code>\t</code> Horizontaler Tab</li><li>▷ <code>\b</code> Backspace</li><li>▷ <code>\n</code> Neue Zeile</li><li>▷ Auch Darstellung im Hexacode (<code>\u0039A</code>)</li></ul>
Enumeration	<ul style="list-style-type: none"><li>▷ Zusammenfassung mehrerer Konstanten (feste Anzahl)</li><li>▷ Erzeugung meist in eigener <code>.java</code> Datei</li><li>▷ <code>enum MyDirection {DOWN, RIGHT}</code></li><li>▷ Keine Objekterzeugung von Enumeration möglich</li><li>▷ Abspeichern in Variable des Enum-Typs ist jedoch möglich</li><li>▷ <code>MyDirection dir = MyDirection.DOWN;</code></li><li>▷ Klassenmethoden:<ul style="list-style-type: none"><li>◊ <code>values()</code> // Returns array with all enum components</li><li>◊ <code>name()</code> // Returns the name of the calling object as string</li></ul></li></ul>
Referenztypen	<ul style="list-style-type: none"><li>▷ Alle Typen, die keine primitiven Datentypen sind</li><li>▷ Unterscheidung zwischen Referenz und eigentlichem Objekt</li><li>▷ Gleichheitsoperator <code>==</code> vergleicht nur die Referenz (Objektidentität)<ul style="list-style-type: none"><li>◊ Verweis auf dasselbe Objekt</li></ul></li><li>▷ Wertgleichheit bezieht sich auf das Objekt an sich<ul style="list-style-type: none"><li>◊ Deep Copy ⇒ An allen parallelen Stellen Wertgleichheit</li><li>◊ Shallow Copy ⇒ Nur Kopie der Adressen</li></ul></li><li>▷ Ohne Initialisierung: <code>Null</code></li></ul>

## 5 Exceptions (java.lang.Exception;)

Exception-Klassen	<ul style="list-style-type: none"> <li>▷ Alle Klassen, die direkt oder indirekt von java.lang.Exception abgeleitet sind</li> </ul>
Exception werfen	<ul style="list-style-type: none"> <li>▷ <code>throws Exception {...}</code> nach Parameterliste im Methodenkopf</li> <li>▷ Dies signalisiert, dass die Methode mindestens einen Fehler wirft</li> <li>▷ Die geworfene Exception muss vom <code>throws</code>-Typ oder Subtyp sein</li> <li>▷ Auch mehrere Exceptions möglich, mit einem Komma getrennt</li> <li>▷ Werfen der Exception: <ul style="list-style-type: none"> <li>◊ z.B.: <code>throw new Exception (No lower case letter!);</code></li> <li>◊ Hier wird als Parameter für die Objekterstellung ein String übergeben</li> </ul> </li> <li>▷ <code>throws</code>: <ul style="list-style-type: none"> <li>◊ Führt zur Beendigung der Methode</li> <li>◊ Liefert das geworfene Exception-Objekt zurück</li> </ul> </li> </ul>
Exception fangen	<ul style="list-style-type: none"> <li>▷ Bei Methoden, die Exceptions werfen, wird ein <code>try-catch</code>-Block benötigt</li> <li>▷ Aufbau: <ul style="list-style-type: none"> <li>◊ Methoden, die Exceptions werfen in <code>try {...}</code> aufrufen</li> <li>◊ Falls Exception auftritt wird <code>catch (Exception exc) {...}</code> aufgerufen</li> <li>◊ <code>catch</code> muss direkt im Anschluss nach <code>try</code> stehen</li> <li>◊ Falls kein Fehler auftritt, wird <code>catch</code> übersprungen</li> <li>◊ Das Programm wird dann normal weiter ausgeführt</li> </ul> </li> <li>▷ Es sind auch mehrere <code>catch</code>-Blöcke mit versch. Parametern möglich</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getMessage(); // Returns the error message as a string</code></li> <li>◊ <code>printStackTrace(); // Ausgabe des Call-Stacks</code></li> </ul> </li> <li>▷ Alle möglichen Exceptions müssen durch den <code>catch</code>-Block abgedeckt sein</li> <li>▷ Falls Exception zu mehreren <code>catch</code>-Blöcken 'passt', wird der Erste ausgeführt <ul style="list-style-type: none"> <li>◊ Deswegen Reihung der <code>catch</code>-Blöcke von Subtyp nach Supertyp</li> </ul> </li> <li>▷ Auch mehrere Exceptions in einem <code>catch</code>-Block möglich mit <code>  </code></li> </ul>
Weiterreichen	<ul style="list-style-type: none"> <li>▷ Weiterreichen der Fehlermeldung durch <code>throws</code> im Methodenkopf möglich</li> <li>▷ Kein <code>try-catch</code>-Block notwendig</li> <li>▷ Main-Methode kann z.B. keine Exceptions weiterreichen</li> </ul>
<code>try-with-resources</code>	<ul style="list-style-type: none"> <li>▷ Für Ressourcen, die unbedingt wieder geschlossen werden müssen</li> <li>▷ Öffnung der Ressource in runden Klammern: <code>try (Printer p =... ) {...}</code></li> <li>▷ Mehrere Ressourcen möglich, getrennt durch Semikolon</li> </ul>
Runtime Exceptions	<ul style="list-style-type: none"> <li>▷ Ausnahme zu <code>try</code>-Blöcken</li> <li>▷ Exceptions von java.lang.RuntimeException und Subtypen</li> <li>▷ z.B.: <code>IndexOutOfBoundsException</code>, <code>NullPointerException</code></li> <li>▷ Grund: Vermeidung von dauerenden <code>try</code>-Blöcken</li> </ul>
Throwable und Error	<ul style="list-style-type: none"> <li>▷ Exception und Error sind beide von Throwable abgeleitet</li> <li>▷ Alle drei befinden sich im Paket java.lang</li> <li>▷ Error: <ul style="list-style-type: none"> <li>◊ Werden geworfen, falls Fehlerbehandlung keinen Sinn macht</li> <li>◊ Programmabbruch als Ausweg</li> </ul> </li> <li>▷ <code>AssertionError</code>: <ul style="list-style-type: none"> <li>◊ <code>throw new AssertionError("Bad!");</code></li> <li>◊ Kurzform: <code>assert x == 2: "Bad!";</code></li> <li>◊ <b>Wichtig:</b> Bedingung muss negiert werden!</li> <li>◊ Assertanweisungen sinnvoll, da kurz und übersichtlich</li> <li>◊ Können zusätzlich vom Compiler an- und abgeschaltet werden</li> <li>◊ z.B.: Verwendung für Tests für Methoden und späteres Abschalten</li> </ul> </li> <li>▷ Solche Tests werden White-Box-Tests genannt</li> </ul>



## 6 Fehler

Kompilierzeitfehler (compile-time errors)	<ul style="list-style-type: none"> <li>▷ Falsche Klammersetzung, falsche Schlüsselwörter,..</li> <li>▷ Programm wird nicht übersetzt ⇒ Fehlermeldung vom Compiler</li> </ul>
Laufzeitfehler (run-time errors)	<ul style="list-style-type: none"> <li>▷ Tritt während der Ausführung auf</li> <li>▷ Führt zum Abbruch des Programms ⇒ Ausgabe der Fehlermeldung</li> <li>▷ Kann nicht vom Compiler entdeckt werden</li> <li>▷ IndexOutOfBounds, NullPointerException,..</li> </ul>

## 7 Files

System Properties (java.lang.System)	<ul style="list-style-type: none"> <li>▷ Attribute der Umgebung, in denen das Java Programm abläuft</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getProperty</code> <ul style="list-style-type: none"> <li>- Erhält <code>String</code> und gibt <code>String</code> zurück</li> </ul> </li> <li>◊ z.B.: <code>String homeDir = System.getProperty("user.home");</code></li> <li>◊ Mögliche Strings: <ul style="list-style-type: none"> <li>- <code>"user.home"</code> // Home directory</li> <li>- <code>"user.dir"</code> // Working directory</li> <li>- <code>"user.name"</code> // Account name</li> <li>- <code>"file.separator"</code> // Zeichen zur Dateitrennung</li> <li>- <code>"line.separator"</code> // Zeichen zur Zeilentrennung</li> </ul> </li> </ul> </li> <li>▷ <code>System.out</code>: <ul style="list-style-type: none"> <li>◊ Klassenattribut <code>out</code> von <code>System</code> ist von Klasse <code>PrintStream</code></li> <li>◊ <code>PrintStream</code> hat also auch Methoden wie <code>println</code></li> </ul> </li> <li>▷ <code>System.err</code>: <ul style="list-style-type: none"> <li>◊ Auch <code>err</code> ist von Klasse <code>PrintStream</code></li> <li>◊ Hierhin werden die Fehlerausgaben geschrieben</li> <li>◊ z.B. sinnvoll um Fehler in separate Log-Datei umzuleiten</li> </ul> </li> <li>▷ <code>System.in</code>: <ul style="list-style-type: none"> <li>◊ Auch <code>in</code> ist von Klasse <code>PrintStream</code></li> <li>◊ Liest Tastatureingaben</li> </ul> </li> <li>▷ Diese drei Attribute können auch auf andere Streams gesetzt werden <ul style="list-style-type: none"> <li>◊ z.B.: andere <code>FileInputStreams/FileOutputStreams</code></li> <li>◊ <code>System.setIn(in); System.setOut(out); System.setErr(err);</code></li> </ul> </li> </ul>
Klasse Path / Paths	<ul style="list-style-type: none"> <li>▷ Beide in <code>java.nio.file</code></li> <li>▷ Objekt der Klasse <code>Path</code> verwaltet einen Pfadnamen <ul style="list-style-type: none"> <li>◊ Dort muss nicht unbedingt etwas existieren</li> </ul> </li> <li>▷ <code>Paths</code> wird nur dazu genutzt um Objekt von <code>Path</code> zu erzeugen <ul style="list-style-type: none"> <li>◊ z.B.: <code>Path path = Paths.get(homeDir, "fop.txt");</code></li> </ul> </li> </ul>

Klasse Files	<ul style="list-style-type: none"> <li>▷ Aus Package <code>java.nio.file</code></li> <li>▷ Nützliche Sammlung von Klassenmethoden rund um Dateien</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◇ <code>lines</code> // <code>Files.lines(path)</code>; <ul style="list-style-type: none"> <li>- Öffnet Datei an übergebenem Pfad</li> <li>- Liefert einen Stream von Strings, ein String pro Zeile</li> <li>- Zeilenende durch <code>"file.separator"</code> gekennzeichnet</li> <li>- <code>IOException</code>, falls Problem beim Öffnen der Datei (<code>java.io</code>)</li> </ul> </li> <li>◇ <code>exists</code> // <code>Files.exists(path)</code>; <ul style="list-style-type: none"> <li>- <code>true</code>, wenn es dort Datei/Verzeichnis gibt</li> </ul> </li> <li>◇ <code>isReadable(path)</code> <ul style="list-style-type: none"> <li>- Fragt lesende Zugriffsrechte ab</li> </ul> </li> <li>◇ <code>isWritable(path)</code> <ul style="list-style-type: none"> <li>- Fragt schreibende Zugriffsrechte ab</li> </ul> </li> <li>◇ <code>isRegularFile(path)</code> <ul style="list-style-type: none"> <li>- <code>true</code>, falls es eine reguläre Datei ist (kein Verzeichnis)</li> </ul> </li> <li>◇ <code>isDirectory(path)</code> <ul style="list-style-type: none"> <li>- <code>true</code>, falls es ein Verzeichnis ist</li> </ul> </li> <li>◇ <code>size(path)</code> // <code>long size = Files.size(path)</code>; <ul style="list-style-type: none"> <li>- Fragt die Größe der Datei ab</li> <li>- <code>long</code>, da die Dateigröße oft nicht in <code>int</code> passt</li> </ul> </li> <li>◇ <code>createFile(path)</code> <ul style="list-style-type: none"> <li>- Richtet Datei an der übergebenen Stelle ein</li> </ul> </li> <li>◇ <code>copy(path1, path2)</code> <ul style="list-style-type: none"> <li>- Kopieren von Pfad 1 nach Pfad 2</li> </ul> </li> <li>◇ <code>move(path1, path2)</code> <ul style="list-style-type: none"> <li>- Umbenennen einer Datei, oft auch Bewegen genannt</li> </ul> </li> <li>◇ <code>delete(path)</code> <ul style="list-style-type: none"> <li>- Entfernen einer Datei</li> <li>- <code>NoSuchElementException</code>, falls nicht vorhanden</li> </ul> </li> <li>◇ <code>deleteIfExists(path)</code> <ul style="list-style-type: none"> <li>- Falls das Objekt nicht existiert, passiert garnichts</li> </ul> </li> </ul> </li> </ul>
Beispiel: Einlesen einer Datei in einen String	<pre> 1 String homeDir = System.getProperty("user.home"); 2 Path path = Paths.get(homeDir, "fop", "streams.txt"); 3 try (Stream&lt;String&gt; stream = Files.lines(path)) { 4     String fileContentAsString = stream.reduce(String::concat); 5 } catch (IOException exc) { 6     System.out.print("Could not open file") 7 } </pre> <ul style="list-style-type: none"> <li>▷ <code>try-with-resources</code> wird für Interface <code>AutoCloseable</code> verwendet</li> </ul>
Bytedaten	<ul style="list-style-type: none"> <li>▷ Direkt, ohne Bezug zu Streams</li> <li>▷ Klassen und Interfaces finden sich in <code>java.io</code></li> <li>▷ Byteweise Verarbeitung sinnvoll für Audio oder Bilddateien, nicht für Text</li> <li>▷ Wird aber meist durch Bibliotheken oder Ähnliches gehandhabt</li> </ul>
Bytedaten lesen	<ul style="list-style-type: none"> <li>▷ Verwendung eines <code>InputStream</code>-Objekts</li> <li>▷ <code>InputStream</code> abstrakt, deswegen nur Subtypen z.B.: <code>FileInputStream</code> <ul style="list-style-type: none"> <li>◇ <code>FileInputStream</code> nutzt den Namen der Datei als String im Konstruktor</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◇ <code>read()</code> <ul style="list-style-type: none"> <li>- Liest nächstes Byte in ein <code>int</code></li> <li>- Überprüfung, ob -1 um zu prüfen, ob Dateiende erreicht ist</li> </ul> </li> </ul> </li> <li>▷ Beispiel: <pre> 1 FileInputStream in = new FileInputStream (fileName); 2 int n = in.read(); 3 if (n == 1) return; </pre> </li> </ul>

Bytedaten schreiben	<ul style="list-style-type: none"> <li>▷ Verwendung eines <code>OutputStream</code>-Objekts</li> <li>▷ <code>OutputStream</code> abstrakt, deswegen nur Subtypen z.B.: <code>FileOutputStream</code> <ul style="list-style-type: none"> <li>◊ <code>FileOutputStream</code> nutzt den Namen der Datei als String im Konstruktor</li> <li>◊ Existiert die Datei schon, geht der Inhalt verloren</li> <li>◊ Existiert die Datei nicht, wird sie erstellt</li> <li>◊ Zweiter Konstruktor mit <code>boolean</code> als zweiten Parameter: <ul style="list-style-type: none"> <li>- Falls <code>false</code>: Verhält sich wie normaler Konstruktor</li> <li>- Falls <code>true</code>: Inhalt geht nicht verloren, wird hinten angehängen</li> </ul> </li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>write()</code></li> <li>◊ Hat <code>int</code> als formalen Parametertyp</li> <li>◊ Schreibt nur unterstes Byte dieses <code>int</code></li> </ul> </li> <li>▷ Beispiel: <pre>1  FileOutputStream out = new FileOutputStream(fileName); 2  int i = 5; 3  out.write(i);</pre> </li> </ul>
Relevante Subtypen von Input-/OutputStream	<ul style="list-style-type: none"> <li>▷ Geschwindigkeit beim Lesen/Schreiben ist relevant</li> <li>▷ <code>BufferedInputStream</code>: <ul style="list-style-type: none"> <li>◊ liest mehrere Bytes auf einmal ein</li> <li>◊ Konstruktor: <code>BufferedInputStream(InputStream in)</code></li> <li>◊ Verwendet im Konstruktor z.B. einen <code>FileInputStream</code></li> </ul> </li> <li>▷ <code>BufferedOutputStream</code>: <ul style="list-style-type: none"> <li>◊ Schreibt zuerst in internen Puffer</li> <li>◊ Falls dieser voll ist, wird in die Datei geschrieben</li> <li>◊ Konstruktor: <code>BufferedOutputStream(OutputStream out)</code></li> <li>◊ Schreibt die Daten auf den <code>OutputStream</code> im Parameter</li> </ul> </li> <li>▷ <code>PrintStream</code>: <ul style="list-style-type: none"> <li>◊ Ersatz für <code>OutputStream</code> im Package <code>java.io</code></li> <li>◊ Konstruktor: <code>PrintStream(OutputStream out)</code></li> <li>◊ Dient als Konvertierer von primitiven Datentypen und String in die byteweise Darstellung</li> <li>◊ Das eigentliche Schreiben übernimmt der übergebene <code>OutputStream</code></li> <li>◊ Methode <code>print</code> <ul style="list-style-type: none"> <li>- z.B.: <code>out1.print(pi = " ); out1.print(3.14);</code></li> <li>- Byteweise Ausgabe von übergebenen Werten</li> </ul> </li> <li>◊ <code>System.out.print()</code>: <code>out</code> ist von Klasse <code>PrintStream</code></li> <li>◊ Methode <code>println</code> <ul style="list-style-type: none"> <li>- Ausgabe von Werten mit Zeilenumbruch</li> </ul> </li> </ul> </li> </ul>
Mehr Subtypen von Input-/OutputStream	<ul style="list-style-type: none"> <li>▷ <code>java.util.zip.ZipInputStream</code> <ul style="list-style-type: none"> <li>◊ Zum Einlesen von komprimierten Zip-Dateien</li> </ul> </li> <li>▷ <code>java.util.jar.JarInputStream</code> <ul style="list-style-type: none"> <li>◊ Zum Einlesen von Jar-Dateien</li> <li>◊ Jar-Dateien enthalten kompilierte Java-Dateien, mit zip komprimiert</li> </ul> </li> <li>▷ <code>javax.sound.sampled.AudioInputStream</code> <ul style="list-style-type: none"> <li>◊ für Audio-Dateien</li> </ul> </li> <li>▷ <code>java.io.PipedInputStream</code> / <code>java.io.PipedOutputStream</code> <ul style="list-style-type: none"> <li>◊ Zwei aneinander gekoppelte Lese/Schreib-Klassen</li> </ul> </li> </ul>
Textdaten direkt	<ul style="list-style-type: none"> <li>▷ Bequemere Zugriffsmöglichkeiten für Textdaten vorhanden</li> <li>▷ <code>Reader</code> und <code>Writer</code> aus Package <code>java.io</code></li> <li>▷ Textdatei besteht aus einzelnen Zeichen aka <code>char</code> <ul style="list-style-type: none"> <li>◊ Jedes <code>char</code> ist zwei Byte groß</li> </ul> </li> </ul>

Textdaten lesen	<ul style="list-style-type: none"> <li>▷ Komplette analog zu <code>InputStream</code> und <code>FileInputStream</code></li> <li>▷ <code>Reader</code> abstrakt, deswegen nur Subtypen z.B. <code>FileReader</code> <ul style="list-style-type: none"> <li>◊ <code>FileReader</code> nutzt den Namen der Datei als String im Konstruktor</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>read</code> <ul style="list-style-type: none"> <li>- Liest <code>char</code>-Werte ein</li> <li>- Verschiedene Implementationen z.B.: kein Parameter → einzelner <code>char</code></li> <li>- Mit <code>char</code>-Array: Liest so viele ein, bis Array voll ist</li> </ul> </li> </ul> </li> <li>▷ Beispiel: <pre> 1  FileReader reader1 = new FileReader(fileName); 2  char[] buffer = new char[256]; 3  int n = reader1.read(buffer); 4  // n ist in diesem Fall die Anzahl der gelesenen chars </pre> </li> <li>▷ <code>BufferedReader</code> <ul style="list-style-type: none"> <li>◊ Konstruktor: <code>BufferedReader(Reader in)</code></li> <li>◊ Methode <code>readLine()</code>; <ul style="list-style-type: none"> <li>- Liest alles vom letzten gelesenen Zeichen bis zum Zeilenende</li> <li>- Also meist eine ganze Zeile</li> </ul> </li> </ul> </li> <li>▷ Verknüpfung mit byteweisem Einlesen: <ul style="list-style-type: none"> <li>◊ evtl. sinnvoll, falls offener <code>InputStream</code> auf Text-Datenquelle</li> <li>◊ Die Brücke bildet hier der Subtyp <code>InputStreamReader</code> <pre> 1  InputStream in = ...; 2  Reader reader = new InputStreamReader(in); </pre> </li> </ul> </li> </ul>
Textdaten schreiben	<ul style="list-style-type: none"> <li>▷ <code>Writer</code> abstrakt, deswegen nur Subtypen z.B. <code>FileWriter</code> <ul style="list-style-type: none"> <li>◊ <code>FileWriter</code> benutzt den Namen der Datei als String im Konstruktor</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>write</code> <ul style="list-style-type: none"> <li>- Schreibt einzelnen <code>char</code> oder ganzen String</li> </ul> </li> </ul> </li> <li>▷ Beispiel: <pre> 1  FileWriter writer1 = new FileWriter(fileName); 2  writer1.write('H'); 3  writer1.write("ello World"); </pre> </li> <li>▷ Verknüpfung mit byteweisem Schreiben: <ul style="list-style-type: none"> <li>◊ Die Brücke bildet hier der Subtyp <code>OutputStreamWriter</code> <pre> 1  OutputStream out = ...; 2  Writer writer = new OutputStreamWriter(out); </pre> </li> </ul> </li> </ul>

## 8 Functional Interfaces und Lambda-Ausdrücke

Functional Interface	<ul style="list-style-type: none"> <li>▷ Interface, bei dem genau eine Methode weder <b>default</b> oder <b>static</b> ist <ul style="list-style-type: none"> <li>◊ Diese Methode heißt funktionale Methode dieses <b>Functional Interface</b></li> </ul> </li> <li>▷ Jedes <b>Functional Interface</b> hat genau eine funktionale Methode</li> <li>▷ <b>Functional Interface</b> <code>IntToDoubleFunction</code> <ul style="list-style-type: none"> <li>◊ Aus Package <code>java.util.function</code></li> <li>◊ Hat die funktionale Methode <code>double applyAsDouble(int n);</code></li> </ul> </li> </ul>
Lambda-Ausdrücke Einführung	<ul style="list-style-type: none"> <li>▷ Sind Literale von Funktionstypen</li> <li>▷ Abgekürzte Schreibweise für den Aufruf der Hauptmethode eines Func. Interface</li> <li>▷ Verwendung am Beispiel <code>IntToDoubleFunction</code> <b>ohne</b> Lambda: <pre>IntToDoubleFunction fct1 = new X(2); double y = fct1.applyAsDouble(10);</pre> <ul style="list-style-type: none"> <li>- X ist hier eine Klasse, die das Interface und die Methode implementiert</li> </ul> </li> <li>▷ Verwendung am Beispiel <code>IntToDoubleFunction</code> <b>mit</b> Lambda: <pre>IntToDoubleFunction fct2 = x -&gt; x * 10; double z = fct2.applyAsDouble(10);</pre> <ul style="list-style-type: none"> <li>- Richtet nicht sichtbare Klasse ein, die Interface implementiert</li> <li>- Lambda-Ausdruck wird für die funktionale Methode verwendet</li> <li>- Speichern der Referenz eines Objektes dieser Klasse in <code>fct2</code></li> </ul> </li> </ul>
Closure	<ul style="list-style-type: none"> <li>▷ Falls der Parameter (oben 10) zur Laufzeit nicht feststeht (z.B. y): <ul style="list-style-type: none"> <li>◊ Unsichtbare Klasse erhält Attribut, das durch Konstruktor initialisiert wird</li> <li>◊ Verwendung dieses Attributs innerhalb der Klasse</li> </ul> </li> <li>▷ Info aus Entstehungskontext des Lambda-Ausdrucks wird mitgespeichert <ul style="list-style-type: none"> <li>◊ Aktueller Wert aber nicht unbedingt kopiert, sondern referenziert</li> <li>◊ Vorsicht bei Änderungen</li> </ul> </li> </ul>
Lambda-Ausdrücke Aufbau	<ul style="list-style-type: none"> <li>▷ <code>n -&gt; n % 2 == 1</code> <ul style="list-style-type: none"> <li>◊ Kurzform</li> </ul> </li> <li>▷ <code>(int n) -&gt; {return n % 2 == 1;}</code> <ul style="list-style-type: none"> <li>◊ richtige lange Form, Typangabe des Parameters optional</li> </ul> </li> <li>▷ <code>(int n, double x) -&gt; {System.out.print(x); System.out.print(n);}</code> <ul style="list-style-type: none"> <li>◊ Langform ermöglicht mehrere Anweisungen</li> </ul> </li> </ul>
Beispiel Prädikate	<ul style="list-style-type: none"> <li>▷ Prädikat: boolsche Funktionen, die entweder <b>true</b> oder <b>false</b> zurückliefert</li> <li>▷ Gut für Methoden, die z.B. variablen Filter implementieren</li> <li>▷ <code>java.util.function.IntPredicate</code>: <ul style="list-style-type: none"> <li>◊ Funktionale Methode <code>boolean test(int x);</code></li> <li>◊ Beispiel: <ul style="list-style-type: none"> <li>- <code>IntPredicate pred1 = n -&gt; n % 2 == 1</code></li> <li>- Gibt <b>true</b> zurück, falls <code>n</code> ungerade ist</li> </ul> </li> </ul> </li> <li>▷ <code>java.util.function.IntPredicate</code> hat noch <b>default</b>-Methoden: <ul style="list-style-type: none"> <li>◊ Zugriff auf diese über <code>pred1</code>: <ul style="list-style-type: none"> <li>- <code>IntPredicate pred4 = pred1.negate();</code></li> <li>- Die Klasse des Objekts <code>pred1</code> implementiert ja das Interface</li> </ul> </li> <li>◊ Ergänzung des <b>Functional Interface</b> durch diese <b>default</b>-Methoden</li> <li>◊ <code>IntPredicate[] predicates = new IntPredicate[6];</code></li> <li>◊ <code>predicates[0] = n -&gt; n &gt; 0;</code></li> </ul> </li> </ul>

## 9 Graphical User Interface

Window Manager	<ul style="list-style-type: none"> <li>▷ Systemprozess, der permanent im Hintergrund als <b>Service</b> läuft</li> <li>▷ Stellt generelle, anwendungsunspezifische Funktionalitäten zur Verfügung <ul style="list-style-type: none"> <li>◊ Öffnen, Schließen, Ikonifizieren, Größe ändern</li> <li>◊ Rahmen um Fenster, Bildschirmhintergrund</li> </ul> </li> </ul>
----------------	--

Klasse Frame	<ul style="list-style-type: none"> <li>▷ Abgeleitet von <code>java.awt.Window</code>; (awt = abstract window toolkit)</li> <li>▷ Im Gegensatz zu <code>Window</code> aber mit Rahmen (vom <code>Window Manager</code> verwaltet)</li> <li>▷ Beispielkonstruktor: <code>Frame frame = new Frame(string); // Fenstertitel</code></li> <li>▷ Vorhergehensweise: <ul style="list-style-type: none"> <li>◊ Erstellung einer Klasse, die <code>Frame</code> erweitert</li> <li>◊ Hinzufügen von Funktionalitäten</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>setVisible(boolean b)</code> <ul style="list-style-type: none"> <li>- <code>Frame</code> ist entweder sichtbar oder unsichtbar</li> <li>- Standardmäßig unsichtbar</li> <li>- Erst Fenster aufbauen, dann sichtbar machen</li> </ul> </li> <li>◊ <code>setBackground(Color bgColor)</code> <ul style="list-style-type: none"> <li>- Setzt die Hintergrundfarbe des Fensters</li> </ul> </li> <li>◊ <code>dispose()</code> <ul style="list-style-type: none"> <li>- Alle Ressourcen des Fensters und der Bestandteile werden freigegeben</li> </ul> </li> <li>◊ <code>setExtendedState(int state)</code> <ul style="list-style-type: none"> <li>- Setzt den Status des Fensters</li> <li>- <code>ICONIFIED</code>: Ikonifiziert das Fenster</li> <li>- <code>NORMAL</code>: Deikonifiziert das Fenster</li> <li>- <code>MAXIMIZED_HORIZ</code>: Ausbreitung auf gesamte Horizontale</li> </ul> </li> <li>◊ <code>add(Component comp)</code> <ul style="list-style-type: none"> <li>- Fügt den übergebenen Komponenten zum <code>Frame</code> hinzu</li> </ul> </li> </ul> </li> </ul>
Komponenten	<ul style="list-style-type: none"> <li>▷ Eigene Klasse für jede Komponente</li> <li>▷ Alle Klassen oder Interfaces aus <code>java.awt</code>, falls nicht anders gesagt</li> <li>▷ Werden mithilfe von <code>add(Component comp)</code> zum Fenster hinzugefügt</li> </ul>
Button	<ul style="list-style-type: none"> <li>▷ Konstruktor: <code>Button(String label) // Text auf dem Button</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>setFont(Font f)</code> <ul style="list-style-type: none"> <li>- Zum Setzen der Schriftart</li> <li>- Konstruktor <code>Font: Font(String name, int style, int size)</code></li> </ul> </li> <li>◊ <code>addActionListener(ActionListener l)</code> <ul style="list-style-type: none"> <li>- Fügt den übergebenen <code>ActionListener</code> hinzu</li> <li>- Bei jedem Klick wird <code>actionPerformed</code> des <code>Listeners</code> aufgerufen</li> <li>- Auch mehrere möglich</li> <li>- Automatische Einrichtung des <code>Event Dispatch Thread</code></li> </ul> </li> <li>◊ <code>setLabel(String label)</code> <ul style="list-style-type: none"> <li>- Setzt den Titel des <code>Button</code></li> </ul> </li> </ul> </li> </ul>
Interface ActionListener	<ul style="list-style-type: none"> <li>▷ Zugehörig zu <code>Button</code></li> <li>▷ Aus Package <code>java.awt.event</code></li> <li>▷ Funktionales Interface</li> <li>▷ Funktionale Methode <code>actionPerformed (ActionEvent event)</code></li> <li>▷ Vorhergehensweise: <ul style="list-style-type: none"> <li>◊ Erstellen einer eigenen Klasse, die <code>ActionListener</code> implementiert</li> <li>◊ Erstellen relevanter Attribute und Konstruktor für gegebenen Fall</li> <li>◊ Implementieren der Methode <code>actionPerformed (ActionEvent event)</code></li> <li>◊ Erstellen eines Objekts unserer Klasse <ul style="list-style-type: none"> <li>- <code>ActionListener listener = new MyListener(frame);</code></li> </ul> </li> <li>◊ Hinzufügen des <code>Listener</code> zum <code>Button</code> <ul style="list-style-type: none"> <li>- <code>button.addActionListener(listener);</code></li> </ul> </li> </ul> </li> <li>▷ Alternativ: <ul style="list-style-type: none"> <li>◊ Erstellung des <code>Listener</code> in der Subklasse des <code>Frame</code> <ul style="list-style-type: none"> <li>- Keine <code>Frame</code>-Übergabe notwendig</li> <li>- z.B.: als <code>private</code>-Klasse (Stichwort: <code>nested classes</code>)</li> </ul> </li> </ul> </li> </ul>

Klasse <code>ActionEvent</code>	<ul style="list-style-type: none"> <li>▷ Übergebener Parameter bei <code>actionPerformed</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getWhen()</code> <ul style="list-style-type: none"> <li>- Gibt die Uhrzeit des Geschehnisses als <code>long</code> zurück</li> <li>- Nützlich: <code>java.sql.Timestamp</code></li> <li>- <code>Timestamp stamp = new Timestamp (event.getWhen());</code></li> <li>- Methoden: <code>stamp.getHour(); stamp.getMinute();</code></li> </ul> </li> </ul> </li> </ul>
Übersicht Listener und Events	<ul style="list-style-type: none"> <li>▷ <code>Listener</code>-Interface ↔ <code>Event</code>-Klasse</li> <li>▷ <code>KeyListener</code> ↔ <code>KeyEvent</code></li> <li>▷ <code>MouseListener</code> ↔ <code>MouseEvent</code></li> <li>▷ <code>MouseMotionListener</code> ↔ <code>MouseEvent</code></li> <li>▷ <code>MouseWheelListener</code> ↔ <code>MouseWheelEvent</code></li> <li>▷ <code>WindowFocusListener</code> ↔ <code>WindowEvent</code></li> <li>▷ <code>WindowListener</code> ↔ <code>WindowEvent</code></li> <li>▷ <code>WindowStateListener</code> ↔ <code>WindowEvent</code></li> <li>▷ Hinzufügen: <ul style="list-style-type: none"> <li>◊ <code>addKeyListener(...)</code></li> <li>◊ <code>addMouseListener(...)</code></li> <li>◊ <code>addWindowListener(...)</code></li> </ul> </li> </ul>
Adapter	<ul style="list-style-type: none"> <li>▷ Verwendung von Adaptern, wenn passendes Interface nicht <b>functional</b> ist <ul style="list-style-type: none"> <li>◊ z.B. Interface <code>KeyListener</code>, <code>MouseListener</code>,...</li> <li>◊ Diese Interfaces besitzen mehrere Methoden</li> </ul> </li> <li>▷ Adapter sind Klassen und bestehen zu jedem <code>Listener</code>-Interface <ul style="list-style-type: none"> <li>◊ z.B.: <code>KeyAdapter</code>, <code>MouseAdapter</code></li> <li>◊ Diese Adapter implementieren das dazugehörige Interface</li> <li>◊ Die Methoden werden jedoch leer gelassen</li> </ul> </li> <li>▷ Vorteil vom Adapter: <ul style="list-style-type: none"> <li>◊ Nicht alle Methoden müssen implementiert werden</li> <li>◊ Nur die genutzten Methoden (z.B.: <code>keyPressed()</code>) werden implementiert</li> </ul> </li> <li>▷ Verwendung: <ul style="list-style-type: none"> <li>◊ Erweitern der eigenen <code>Listener</code>-Klasse mit Adapter</li> <li>◊ z.B.: <code>public class MyKeyListener extends KeyAdapter {...}</code></li> </ul> </li> </ul>
Interface <code>KeyListener</code>	<ul style="list-style-type: none"> <li>▷ Abhören der Tastatur</li> <li>▷ Erstellen eigener Klasse, die die Klasse <code>KeyAdapter</code> (siehe Adapter) erweitert</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void keyPressed (KeyEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim Herunterdrücken einer Taste ausgeführt</li> </ul> </li> <li>◊ <code>public void keyReleased (KeyEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim Loslassen einer Taste ausgeführt</li> </ul> </li> <li>◊ <code>public void keyTyped (KeyEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim Antippen einer Taste ausgeführt</li> </ul> </li> </ul> </li> </ul>

Klasse KeyEvent	<ul style="list-style-type: none"> <li>▷ Übergebener Parameter bei z.B.: <code>keyPressed</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getKeyCode()</code> <ul style="list-style-type: none"> <li>- Liefert die Kodierung der gedrückten Taste zurück</li> </ul> </li> </ul> </li> <li>▷ Klassenkonstanten für jede Taste: <ul style="list-style-type: none"> <li>◊ z.B.: <code>KeyEvent.VK_A</code> // Buchstabe A</li> <li>◊ z.B.: <code>KeyEvent.VK_COLON</code> // Doppelpunkt</li> <li>◊ z.B.: <code>KeyEvent.VK_BACKSPACE</code> // Backspace Taste</li> </ul> </li> <li>▷ Beispiel Verwendung: <pre> 1 public class MyKeyListener extends KeyAdapter { 2     public void keyPressed (KeyEvent event) { 3         switch (event.getKeyCode()) { 4             case KeyEvent.VK_A: ... break; 5             case KeyEvent.VK_COLON: ... break; 6             case KeyEvent.VK_Backspace: ... break; 7         } 8     } 9 }</pre> </li> </ul>
Interface MouseListener	<ul style="list-style-type: none"> <li>▷ Abhören der Maus</li> <li>▷ Erstellen eigener Klasse, die die Klasse <code>MouseAdapter</code> erweitert <ul style="list-style-type: none"> <li>◊ <code>MouseAdapter</code> implementiert alle drei Mouse-Interfaces</li> <li>◊ <code>MouseListener</code>, <code>MouseMotionListener</code>, <code>MouseWheelListener</code></li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void mouseClicked (MouseEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim kurzen Klicken der Maustaste ausgeführt</li> </ul> </li> <li>◊ <code>public void mousePressed (MouseEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim Herunterdrücken der Maustaste ausgeführt</li> </ul> </li> <li>◊ <code>public void mouseReleased (MouseEvent event)</code> <ul style="list-style-type: none"> <li>- Wird beim Loslassen der Maustaste ausgeführt</li> </ul> </li> <li>◊ <code>public void mouseEntered (MouseEvent event)</code> <ul style="list-style-type: none"> <li>- Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich betritt</li> </ul> </li> <li>◊ <code>public void mouseExited (MouseEvent event)</code> <ul style="list-style-type: none"> <li>- Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich verlässt</li> </ul> </li> </ul> </li> </ul>
Interface MouseMotionListener	<ul style="list-style-type: none"> <li>▷ Abhören der Mausbewegung</li> <li>▷ Methoden sind auch in Klasse <code>MouseAdapter</code> enthalten</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void mouseDragged (MouseEvent event)</code></li> <li>◊ <code>public void mouseMoved (MouseEvent event)</code></li> </ul> </li> </ul>
Interface MouseWheelListener	<ul style="list-style-type: none"> <li>▷ Abhören der Mauseingabe</li> <li>▷ Methoden sind auch in Klasse <code>MouseAdapter</code> enthalten</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void mouseWheelMoved (MouseWheelEvent event)</code></li> </ul> </li> </ul>
Klasse MouseEvent	<ul style="list-style-type: none"> <li>▷ Übergebener Parameter bei z.B.: <code>mouseClicked</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getButton()</code> <ul style="list-style-type: none"> <li>- Liefert die gedrückte Taste zurück</li> </ul> </li> <li>◊ <code>getX()</code> <ul style="list-style-type: none"> <li>- Liefert x-Koordinate abhängig vom Ursprung des Bereichs</li> </ul> </li> <li>◊ <code>getY()</code> <ul style="list-style-type: none"> <li>- Liefert y-Koordinate abhängig vom Ursprung des Bereichs</li> </ul> </li> </ul> </li> <li>▷ Klassenkonstanten für Maustasten: <ul style="list-style-type: none"> <li>◊ <code>MouseEvent.BUTTON1</code></li> <li>◊ <code>MouseEvent.BUTTON2</code></li> </ul> </li> </ul>



Klasse MouseEvent	<ul style="list-style-type: none"> <li>▷ Übergebener Parameter bei z.B.: <code>mouseWheelMoved</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getWheelRotation()</code></li> <li>- Liefert die Anzahl der gedrehten Ticks"</li> </ul> </li> </ul>
Interface WindowListener	<ul style="list-style-type: none"> <li>▷ Abhören von Fensteraktionen</li> <li>▷ Erstellen eigener Klasse, die die Klasse <code>WindowAdapter</code> erweitert <ul style="list-style-type: none"> <li>◊ <code>WindowAdapter</code> implementiert alle drei Window-Interfaces</li> <li>◊ <code>WindowListener</code>, <code>WindowStateListener</code>, <code>WindowFocusListener</code></li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void windowOpened (WindowEvent event)</code></li> <li>◊ <code>public void windowClosing (WindowEvent event)</code></li> <li>◊ <code>public void windowClosed (WindowEvent event)</code></li> <li>◊ <code>public void windowClosed (WindowEvent event)</code></li> <li>◊ <code>public void windowDeactivated (WindowEvent event)</code></li> <li>◊ <code>public void windowIconified (WindowEvent event)</code></li> <li>◊ <code>public void windowDeiconified (WindowEvent event)</code></li> </ul> </li> </ul>
Interface WindowStateListener	<ul style="list-style-type: none"> <li>▷ Abhören des Status des Fensters</li> <li>▷ Methoden sind auch in <code>WindowAdapter</code> vorhanden</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void windowStateChanged ( WindowEvent event )</code></li> </ul> </li> </ul>
Interface WindowFocusListener	<ul style="list-style-type: none"> <li>▷ Abhören des Fokus im Bezug auf das Fenster</li> <li>▷ Methoden sind auch in <code>WindowAdapter</code> vorhanden</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>public void windowGainedFocus ( WindowEvent event )</code></li> <li>◊ <code>public void windowLostFocus ( WindowEvent event )</code></li> </ul> </li> </ul>
Klasse Canvas	<ul style="list-style-type: none"> <li>▷ abgegrenzte Zeichenfläche in einem Fenster</li> <li>▷ Vorgehensweise: <ul style="list-style-type: none"> <li>◊ Erstellung eigener Subtyp-Klasse von <code>Canvas</code></li> <li>◊ Implementieren der Methode <code>public void paint (Graphics graphics)</code></li> <li>◊ Füllen der Methode mit eigener Zeichenlogik</li> <li>◊ Verwendung von <code>java.awt.Graphics</code>;</li> <li>◊ Hinzufügen zum <code>Frame</code> mithilfe von <code>add</code></li> </ul> </li> <li>▷ Beleuchtung nützlicher Aspekte von <code>Graphics</code>:</li> <li>▷ <b>FontMetrics</b> <ul style="list-style-type: none"> <li>◊ Informationen über festgelegte Schriftart und Schriftgröße</li> <li>◊ Abfrage: <ul style="list-style-type: none"> <li>- <code>FontMetrics fontM = graphics.getFontMetrics();</code></li> </ul> </li> <li>◊ Abfrage der maximalen Stringhöhe: <ul style="list-style-type: none"> <li>- <code>int maxHeight = fontM.getMaxAscent() + fontM.getMaxDescent();</code></li> <li>- Methoden geben maximalen Abstand von der Basislinie des Textes an</li> </ul> </li> <li>◊ Abfrage der Stringbreite von gegebenem String: <ul style="list-style-type: none"> <li>- <code>int widthStr = fontMetrics.stringWidth(string);</code></li> </ul> </li> </ul> </li> <li>▷ Abfrage des Zeichenfensters als Rechteck: <ul style="list-style-type: none"> <li>- <code>Rectangle area = graphics.getClipBounds();</code></li> <li>- <code>x</code> und <code>y</code> geben den Ursprung an</li> <li>- <code>width</code> und <code>height</code> die Breite und Höhe</li> </ul> </li> <li>▷ Einige Methoden von <code>Graphics</code> <ul style="list-style-type: none"> <li>◊ <code>setColor(Color color)</code></li> <li>◊ <code>fillOval(...)</code></li> <li>◊ <code>drawOval(...)</code></li> <li>◊ <code>drawString(...)</code></li> </ul> </li> </ul>

Klasse Checkbox	<ul style="list-style-type: none"> <li>▷ Kleiner Button (Pin) mit etwas Text</li> <li>▷ Zwei Zustände: An oder Aus</li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>Checkbox(String label)</code> // Titel der Checkbox</li> <li>◊ <code>Checkbox</code> standardmäßig aus</li> </ul> </li> <li>▷ Benötigt ein Objekt vom Typ <code>ItemListener</code> (siehe unten) <ul style="list-style-type: none"> <li>◊ <code>ItemListener item = new MyItemListener(checkbox,...);</code></li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>isSelected()</code> <ul style="list-style-type: none"> <li>- <code>true</code>, wenn die <code>Checkbox</code> an ist</li> </ul> </li> <li>◊ <code>setLabel(string);</code> <ul style="list-style-type: none"> <li>- Setzt den Titel der <code>Checkbox</code></li> </ul> </li> </ul> </li> </ul>
Interface ItemListener	<ul style="list-style-type: none"> <li>▷ Verwendung bei <code>Checkbox</code> und <code>Choice</code></li> <li>▷ Funktionales Interface</li> <li>▷ Funktionale Methode <code>itemStateChanged (ItemEvent event)</code></li> <li>▷ Vorhergehensweise analog zu <code>ActionListener</code> <ul style="list-style-type: none"> <li>◊ Erstellung neuer Klasse, die <code>ItemListener</code> implementiert</li> </ul> </li> </ul>
Klasse Choice	<ul style="list-style-type: none"> <li>▷ Repräsentiert ein Auswahlmenü</li> <li>▷ Verwendet auch das Interface <code>ItemListener</code></li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>Choice choice = new Choice();</code></li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>add(string)</code> <ul style="list-style-type: none"> <li>- Hinzufügen neuer Auswahlen</li> <li>- Startet bei Index 0</li> </ul> </li> <li>◊ <code>select(int)</code> <ul style="list-style-type: none"> <li>- Legt eine Auswahl als Standard fest</li> <li>- Übergabe des Index als <code>int</code>     ◊ <code>getSelectedItem()</code></li> <li>- Liefert den ausgewählten String zurück</li> </ul> </li> <li>◊ <code>getSelectedIndex()</code> <ul style="list-style-type: none"> <li>- Liefert Index der aktiven Auswahl</li> </ul> </li> </ul> </li> </ul>
Klasse Label	<ul style="list-style-type: none"> <li>▷ Nicht durch User interagierbares Rechteck mit Text</li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>Label(String text)</code> // Labeltext</li> </ul> </li> <li>▷ Wartet auf Events bei anderen Entitäten</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>setAlignment(int alignment)</code> <ul style="list-style-type: none"> <li>- Auswahl der Zentrierung des Textes</li> <li>- Paramter: <code>Label.CENTER</code>, <code>Label.RIGHT</code>, <code>Label.LEFT</code></li> </ul> </li> <li>◊ <code>setBackground(Color c)</code> <ul style="list-style-type: none"> <li>- Setzen der Hintergrundfarbe</li> </ul> </li> <li>◊ <code>setText(String text)</code> <ul style="list-style-type: none"> <li>- Setzt den Text des Label</li> <li>- z.B.: Aufruf beim Drücken eines Button</li> </ul> </li> </ul> </li> </ul>

Klasse List	<ul style="list-style-type: none"> <li>▷ Auswahlmenü</li> <li>▷ Aus <code>java.awt</code>, <b>nicht</b> <code>java.util</code></li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>List(int rows, boolean multipleMode)</code></li> <li>◊ <code>rows</code> gibt die maximale Anzahl der zugleich angezeigten Menüpunkte an</li> <li>◊ Anzahl der Möglichkeiten größer als <code>rows</code> → Scrollbar</li> <li>◊ <code>multipleMode</code>: Auswahl mehrerer Menüpunkte ermöglichen</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>add(String item)</code> <ul style="list-style-type: none"> <li>- Hinzufügen neuer Menüpunkte</li> </ul> </li> <li>◊ <code>getSelectedIndexes()</code> <ul style="list-style-type: none"> <li>- Liefert die Indizes der ausgewählten Punkte</li> </ul> </li> <li>◊ <code>setMultipleMode(boolean b)</code> <ul style="list-style-type: none"> <li>- De-/Aktivieren der Mehrfachauswahl</li> </ul> </li> </ul> </li> </ul>
Klasse Scrollbar	<ul style="list-style-type: none"> <li>▷ Werden meist automatisch hinzugefügt (<code>List</code>)</li> <li>▷ z.B.: Erstellung eines eigenen Schiebereglers</li> <li>▷ Benötigt ein Objekt vom Typ <code>AdjustmentListener</code> (siehe unten) <ul style="list-style-type: none"> <li>◊ z.B.: <code>AdjustmentListener adjust = new MyAdjustListener(frame);</code></li> </ul> </li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>Scrollbar(int orientation, int value, int visible, int minimum, int maximum)</code></li> <li>◊ <code>orientation</code>: <code>Scrollbar.VERTICAL</code>, <code>Scrollbar.HORIZONTAL</code></li> <li>◊ <code>value</code>: Startwert der Scrollbar</li> <li>◊ <code>visible</code>: Größe des scrollbaren Balkens</li> <li>◊ <code>minimum</code>: Minimal einstellbarer Wert</li> <li>◊ <code>maximum</code>: Maximal einstellbarer Wert</li> </ul> </li> </ul>
Interface AdjustmentListener	<ul style="list-style-type: none"> <li>▷ Verwendung bei <code>Scrollbar</code></li> <li>▷ Funktionales Interface</li> <li>▷ Funktionale Methode: <code>adjustValueChanged (AdjustmentEvent event)</code></li> <li>▷ Vorhergehensweise analog zu <code>ActionListener</code> <ul style="list-style-type: none"> <li>◊ Erstellung neuer Klasse, die <code>AdjustmentListener</code> implementiert</li> </ul> </li> </ul>
Klasse Adjustmentevent	<ul style="list-style-type: none"> <li>▷ Übergebener Parameter bei <code>adjustValueChanged</code></li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getValue()</code> <ul style="list-style-type: none"> <li>- Liefert den neuen Wert der Scrollbar</li> </ul> </li> </ul> </li> </ul>
Klasse Textfield	<ul style="list-style-type: none"> <li>▷ Zeile, vom Nutzer schreibbar</li> <li>▷ z.B.: Benutzername, Passwort, etc..</li> <li>▷ Benötigt ein Objekt vom Typ <code>KeyListener</code> (siehe oben)</li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>TextField(int columns)</code></li> <li>◊ <code>columns</code> gibt die Zeichenzahl in der Zeile an</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>setEchoChar(char c)</code> <ul style="list-style-type: none"> <li>- Anzeige der eingegebenen Zeichen mit anderem Zeichen z.B.: '*'</li> <li>- Rückgängig machen: <code>field.setEchoChar((char) 0);</code></li> </ul> </li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>getText()</code> <ul style="list-style-type: none"> <li>- Liefert den eingegebenen Text als String</li> </ul> </li> </ul> </li> </ul>

Klasse TextArea	<ul style="list-style-type: none"> <li>▷ Eingabebereich über mehrere Zeilen</li> <li>▷ z.B.: Verwendung eines Objekts des Typs <code>FocusListener</code></li> <li>▷ Konstruktor: <ul style="list-style-type: none"> <li>◊ <code>TextArea(String text, int rows, int columns, int scrollbars)</code></li> <li>◊ <code>text</code>: Text, falls Bereich leer und nicht im Mausfokus</li> <li>◊ <code>scrollbars</code>: Legt die Art der Scrollbar fest <ul style="list-style-type: none"> <li>- <code>Scrollbar.BOTH</code>, <code>Scrollbar.HORIZONTAL_ONLY</code></li> <li>- <code>Scrollbar.NONE</code>, <code>Scrollbar.VERTICAL_ONLY</code></li> </ul> </li> <li>◊ <code>rows</code>: Anzahl der Zeilen</li> <li>◊ <code>columns</code>: Breite der Zeilen</li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>setText(String t)</code> <ul style="list-style-type: none"> <li>- Setzt den Text des Textfeldes</li> </ul> </li> <li>◊ <code>getText()</code> <ul style="list-style-type: none"> <li>- Liefert den geschriebenen Text als String</li> </ul> </li> </ul> </li> <li>▷ Leerer Text: <code>("")</code> <ul style="list-style-type: none"> <li>◊ z.B.: Vergleich mit momentanem Text mit <code>equals</code></li> </ul> </li> </ul>
Interface FocusListener	<ul style="list-style-type: none"> <li>▷ Verwendung bei <code>TextArea</code></li> <li>▷ Kein funktionales Interface, trotzdem keine Adapter-Klasse</li> <li>▷ Vorhergehensweise analog zu <code>ActionListener</code></li> <li>▷ Im Gegensatz zu <code>WindowFocusListener</code> auch für einzelne Komponenten</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>focusGained(FocusEvent e)</code> <ul style="list-style-type: none"> <li>- Wird ausgeführt, wenn eine Komponente den Tastaturfokus erhält</li> </ul> </li> <li>◊ <code>focusLost(FocusEvent e)</code> <ul style="list-style-type: none"> <li>- Wird ausgeführt, wenn eine Komponente den Tastaturfokus verliert</li> </ul> </li> </ul> </li> </ul>
Hierarchie graphischer Komponenten	<ul style="list-style-type: none"> <li>▷ Vom <code>java.awt.Component</code> direkt abgeleitet: <ul style="list-style-type: none"> <li>◊ <code>Button</code></li> <li>◊ <code>Canvas</code></li> <li>◊ <code>Checkbox</code></li> <li>◊ <code>Choice</code></li> <li>◊ <code>Label</code></li> <li>◊ <code>List</code></li> <li>◊ <code>Scrollbar</code></li> <li>◊ <code>TextComponent</code> // Supertyp von <code>TextArea</code> und <code>TextField</code></li> <li>◊ <code>Container</code> ▷ Von <code>Container</code> direkt abgeleitet:</li> <li>◊ <code>Window</code></li> </ul> </li> <li>▷ Von <code>Window</code> direkt abgeleitet: <ul style="list-style-type: none"> <li>◊ <code>Frame</code></li> </ul> </li> </ul>
Klasse Component	<ul style="list-style-type: none"> <li>▷ Die meisten Methoden sind hier definiert, aber nicht implementiert <ul style="list-style-type: none"> <li>◊ z.B.: <code>setVisible(boolean b)</code>, <code>setFont(Font f)</code>,...</li> <li>◊ Die Methoden werden in den Komponentenklassen dann implementiert</li> </ul> </li> </ul>

<p>Klasse Container</p>	<ul style="list-style-type: none"> <li>▷ Fasst mehrere Komponenten zu einer zusammen</li> <li>▷ Hinzufügen von <b>Buttons</b>, ..., <b>Windows</b>, <b>Frames</b>, <b>Containern</b> möglich</li> <li>▷ <b>Wichtig:</b> Hinzufügen von <b>Container</b> möglich <ul style="list-style-type: none"> <li>◊ Ähnliche Struktur wie ein Ordnerverzeichnis</li> <li>◊ z.B.: <b>Frame</b> in einem <b>Frame</b></li> </ul> </li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <b>paint</b> (<b>Graphics graphics</b>) <ul style="list-style-type: none"> <li>- In <b>Component</b> definiert, hier überschrieben</li> <li>- Ruft <b>paint</b> für jeden enthaltenen Komponenten auf</li> </ul> </li> <li>◊ <b>add</b> (<b>Component comp</b>) <ul style="list-style-type: none"> <li>- Hinzufügen einer Komponente zum <b>Container</b></li> </ul> </li> <li>◊ <b>add</b> (<b>Component comp</b>, <b>Object constraints</b>) <ul style="list-style-type: none"> <li>- Steuerung der Position mithilfe des zweiten Parameters</li> </ul> </li> <li>- Weiteres bei <b>LayoutManager</b></li> <li>◊ <b>setLayout</b> (<b>LayoutManager manager</b>) <ul style="list-style-type: none"> <li>- Setzen des <b>LayoutManager</b></li> <li>- Dieser steuert die Platzierung der Komponenten</li> <li>- Jeder <b>Container</b> hat zu jedem Zeitpunkt einen <b>LayoutManager</b></li> </ul> </li> <li>◊ <b>validate()</b> <ul style="list-style-type: none"> <li>- Aktualisierung nach z.B.: Größenänderung</li> </ul> </li> </ul> </li> </ul>
<p>Klasse LayoutManager</p>	<ul style="list-style-type: none"> <li>▷ Wird bei Erstellung eines <b>Containers</b> oder Subtyps automatisch eingerichtet <ul style="list-style-type: none"> <li>◊ Standardklasse für für <b>Window</b> und <b>Frame</b> ist <b>BorderLayout</b></li> </ul> </li> <li>▷ <b>BorderLayout</b></li> <li>▷ Einteilung des Fensters in fünf Bereiche <ul style="list-style-type: none"> <li>◊ <b>NORTH</b>, <b>EAST</b>, <b>SOUTH</b>, <b>WEST</b>, <b>CENTER</b></li> <li>◊ Mögliche Positionen als Klassenkonstanten vordefiniert <ul style="list-style-type: none"> <li>- z.B.: <b>BorderLayout.NORTH</b>, <b>BorderLayout.CENTER</b>, ...</li> </ul> </li> <li>◊ Verwendung bei <b>add</b> (<b>Component comp</b>, <b>Object constraints</b>) <ul style="list-style-type: none"> <li>- z.B.: <b>frame.add (comp1, BorderLayout.NORTH)</b>;</li> <li>- Ohne Wahl der Position (normales <b>add</b>): <b>CENTER</b> als Standard</li> </ul> </li> </ul> </li> <li>▷ <b>BorderLayout</b> an sich für das Fenster an sich meist die richtige Wahl <ul style="list-style-type: none"> <li>◊ Aber nicht unbedingt für <b>Container</b> innerhalb eines Fensters</li> </ul> </li> <li>▷ <b>BoxLayout</b> <ul style="list-style-type: none"> <li>◊ Anlegen in einer Reihe nacheinander</li> <li>◊ Wahl ob vertikal oder horizontal im Konstruktor</li> </ul> </li> <li>▷ <b>GridLayout</b> <ul style="list-style-type: none"> <li>◊ Matrixartiges Anlegen (wie Telefonschaltplan)</li> <li>◊ Anzahl Zeilen und Spalten im Konstruktor festgelegt</li> </ul> </li> <li>▷ <b>BorderLayout</b>, <b>BoxLayout</b> und <b>GridLayout</b>: <ul style="list-style-type: none"> <li>◊ Passen Größe der Komponenten anhand der Gesamtsituation an</li> <li>◊ Nicht unbedingt passendste Größe für Komponente</li> </ul> </li> <li>▷ <b>FlowLayout</b> <ul style="list-style-type: none"> <li>◊ Anlegen in einer Zeile nebeneinander <ul style="list-style-type: none"> <li>- Anfangen einer Zeile, falls die alte voll ist</li> </ul> </li> <li>◊ Wählt automatisch die bestmögliche Größe für Komponenten <ul style="list-style-type: none"> <li>- Abfrage über <b>getPreferredSize()</b></li> </ul> </li> </ul> </li> <li>▷ <b>CardLayout</b> <ul style="list-style-type: none"> <li>◊ Zeigt Komponenten nicht alle gleichzeitig, sondern nacheinander</li> <li>◊ Navigation: <b>first</b>, <b>last</b>, <b>next</b>, <b>previous</b></li> </ul> </li> <li>▷ <b>validate()</b> <ul style="list-style-type: none"> <li>◊ Notwendig zur Aktualisierung von sichtbaren Fenstern</li> <li>◊ Wann: <ul style="list-style-type: none"> <li>- Ändern der Anzahl von Komponenten</li> <li>- Ändern der Größe von Komponenten (auch Schrift)</li> </ul> </li> </ul> </li> </ul>

Java Swing  
(javax.swing)

- ▷ `java.awt` als Grundlage
- ▷ Zweite Bibliothek, die die Funktionalitäten erweitert
- ▷ Verbindung zu `java.awt`:
  - ◊ `JFrame` extends `java.awt.Frame`
  - ◊ `JComponent` extends `java.awt.Container`
    - Funktionalitäten von `Container` hier in `JComponent`
- ▷ Von `JComponent` abgeleitet:
  - ◊ `JButton`, `JCheckbox`, `JLabel`
  - ◊ `JList<T>`, `JScrollbar`, `JTextComponent`
- ▷ `JButton`, `JCheckbox` sind aber indirekt abgeleitet:
  - ◊ Zwischenklasse `AbstractButton` bei beiden
    - `public class JButton extends AbstractButton{}`
  - ◊ Bei `JCheckbox` zusätzlich noch `JToggleButton` extends `AbstractButton`
    - `public class JCheckbox extends JToggleButton {}`
  - ◊ **Grund:**
    - Einführung zusätzlicher, eng verwandter, Komponenten
    - Deswegen Auslagerung in Supertyp
- ▷ `JList<T>`:
  - ◊ in `Swing` generisch, Liste von beliebigem Referenztyp
  - ◊ in `java.awt` wird `String` verwendet

## Klasse JComponent

- ▷ Bietet mehr Funktionalitäten als **Component**
- ▷ ToolTips:
  - ◊ Hinzufügen von MouseOver-ToolTips
  - ◊ `setToolTipText(String s)`
    - Setzen des Tooltip-Texts
  - ◊ Deaktivieren mithilfe Übergabe von `null`
- ▷ Randdarstellungen:
  - ◊ Ränder für Komponenten **innerhalb** eines Fensters
  - ◊ `setBorder (Border border)`
  - ◊ Verwendung der Klasse `BorderFactory` zur Erzeugung der `Border`
    - `BorderFactory.createLineBorder(Color color, int thickness)`
      - Simpler rechteckiger Rahmen mit angegebener Dicke und Farbe
    - `BorderFactory.createBevelBorder(BevelBorder.LOWERED)`
    - `BorderFactory.createBevelBorder(BevelBorder.RAISED)`
      - Ganze Komponente mit 3D Effekt (tiefer oder höher erscheinend)
    - `BorderFactory.createEtchedBorder(EtchedBorder.LOWERED)`
    - `BorderFactory.createEtchedBorder(EtchedBorder.RAISED)`
      - Nur Rand mit 3D Effekt (tiefer oder höher erscheinend)
    - `BorderFactory.createEmptyBorder()`
      - Zurücksetzen des Randeffekts
- ▷ Look and Feel:
  - ◊ Anpassung der Gesamterscheinung an Systemstandard
  - ◊ Verwendung von `javax.swing.UIManager`
  - ◊ Setzen des Look and Feel auf Systemstandard:
 

```
1 String s = UIManager.getSystemLookAndFeelClassName();
2 UIManager.setLookAndFeel(s);
```
  - ◊ Setzen des Look and Feel auf z.B. Java-Standard
 

```
Methode: UIManager.setLookAndFeel(LookAndFeel lookAndFeel)
1 UIManager.setLookAndFeel(new MetalLookAndFeel());
```
- ▷ KeyBindings:
  - ◊ Funktionalitäten wie `Listener` automatisch umgesetzt
  - ◊ Erläuterung der Funktion anhand eines Beispiels:
 

```
1 String keyStrokeStr = "alt shift X";
2 KeyStroke keystroke = KeyStroke.getKeyStroke(keyStrokeStr);
3 textArea.getInputMap().put(keystroke, keyStrokeStr);
4 StyledEditorKit.UnderlineAction action
5   = new StyledEditorKit.UnderlineAction();
6 textArea.getActionMap().put(keyStrokeStr, action);
```

    - Zeile 1: Kodierung einer Tastenkombination als String
    - Regeln dafür: Dokumentation `javax.swing.KeyStroke`
    - Zeile 2: Umwandlung des Strings in `KeyStroke`-Objekt
    - Jeder `JComponent` hat `actionMap` und `inputMap` (ähnlich wie `Map`)
    - Zeile 3 und 6: Einfügen von `Key + Value` in jeweilige `Map`
    - Verbindung dieser `Maps` über `Value` von `Input` und `Key` von `Action`
    - Verbindung über `keyStrokeStr` einer Aktion mit Tastenkombination
    - Zeile 4 und 5: Verwendung von `Action extends ActionListener`
    - Verwendung der Methode `actionPerformed`
    - Klassen in Java vorhanden, die `Action` implementieren (`UnderlineAction`)
    - Klasse `UnderlineAction` ist in Klasse `StyledEditorKit` eingebettet
    - Enthält viele Funktionalitäten zum Editieren von Texten
- ▷ Drag & Drop:
  - ◊ Automatisch implementiert, Konfiguration möglich
- ▷ Assistive Technologies:
  - ◊ Unterstützungsmöglichkeiten für Leute mit Handicap
- ▷ Zusätzliche Features von `JFrame`:
  - ◊ Separierung von Hauptmenü und Rest des Fensters

#### Weitere GUI-Klassen

- ▷ **JFormattedTextField:**
  - ◊ Erlaubt Formatierungsregeln für den einzugebenden Text
  - ◊ z.B. für Datumsangaben      ◊ `JFormattedTextField` extends `JTextField`
- ▷ **JPasswordField:**
  - ◊ Zusätzliche Funktionalitäten für Passwörter
- ▷ **JRadioButton:**
  - ◊ kleiner, anklickbarer Bereich
  - ◊ Verwendung im Rahmen eines Objekts von Klasse `ButtonGroup`
  - ◊ Nur ein `RadioButton` in `ButtonGroup` kann gleichzeitig angeklickt sein
- ▷ **JToolBar:**
  - ◊ Vereinfachte Möglichkeit für Standardmenüs
- ▷ **JSlider:**
  - ◊ Klasse für Schieberegler
  - ◊ Besser als Verwendung einer `Scrollbar` als Schieberegler
- ▷ **Popup:**
  - ◊ Popup-Fenster
- ▷ **JTable:**
  - ◊ Umsetzung einer Tabelle
  - ◊ Häufiges Verwendungsbeispiel:

```
1 Object[] [] entries = ...;
2 Object[] columnNames = ...;
3 JTable table = new JTable(entries, columnNames);
4 JScrollPane scrollPane = new JScrollPane(table);
5 table.setFillViewportHeight(true);
6 int[] selectedRows = table.getSelectedRows();
7 int[] selectedColumns = table.getSelectedColumns();
```
  - Zeile 1: Erzeugung der Tabellenmatrix
  - Zeile 2: Erzeugung der Spaltennamen
  - Zeile 3: Konstruktor der Tabelle mit den eben erstellten Arrays
  - Zeile 4: `JScrollPane` kapselt Objekt von `Component` ein
  - Zeigt nur einen Ausschnitt der übergebenen Komponente
  - Fügt außerdem `Scrollbar` ein
  - Zeile 5: Vertikale Streckung der Tabelle, um gesamte Höhe auszufüllen
  - Zeile 6: Abfrage der momentan ausgewählten Zeile
  - Zeile 7: Abfrage der momentan ausgewählten Spalte



## 10 Generics

Wrapper-Klassen	<ul style="list-style-type: none"> <li>▷ primitive Datentypen nicht mit Generizität vereinbar</li> <li>▷ Deswegen benötigen wir eine stellvertretende Klasse → Wrapper-Klassen</li> <li>▷ selber Name, nur mit großem Anfangsbuchstaben (Integer, Long, Character,...)</li> <li>▷ Konstruktor mit Parameter des zugehörigen Datentyps</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>intValue(); // Returns specific value of class</code></li> <li>◊ <code>MAX_VALUE; // Returns max value</code></li> </ul> </li> <li>▷ Boxing/Unboxing: <ul style="list-style-type: none"> <li>◊ Primitiver Datentyp und Wrapper-Klasse sind austauschbar</li> <li>◊ Automatische Umwandlung ineinander</li> <li>◊ Boxing: <code>Integer i = 123;</code></li> <li>◊ Unboxing: <code>System.out.print(i); // 123</code></li> </ul> </li> </ul>
Generische Klassen	<ul style="list-style-type: none"> <li>▷ <code>public class Pair &lt;T1, T2&gt; {...}</code></li> <li>▷ Klasse Pair ist <b>generisch</b> / Klasse Pair ist mit T1 und T2 <b>parametrisiert</b></li> <li>▷ T1 und T2 sind die <b>Typparameter</b> von Klasse Pair</li> <li>▷ T1 und T2 können als Datentypen/Rückgabewerte verwendet werden</li> <li>▷ Können <b>nicht</b> in Klassenmethoden verwendet werden</li> <li>▷ Bei Einrichtung von Objekten von Pair werden die Typparameter festgelegt <ul style="list-style-type: none"> <li>◊ <code>Pair&lt;Integer,Double&gt; pair = new Pair&lt;Integer,Double&gt;(2,3.5);</code></li> <li>◊ Pair ist mit Integer und Double <b>instanziiert</b></li> <li>◊ Typparameter können natürlich auch vom selben Typ sein</li> </ul> </li> </ul>
Generische Methoden	<ul style="list-style-type: none"> <li>▷ Auch in <b>nicht-generischen Klassen</b> generische Methoden möglich</li> <li>▷ <code>public class X {...}</code></li> <li>▷ Einzelne Methode parametrisiert: <ul style="list-style-type: none"> <li>◊ <code>public &lt;T1,T2&gt; Pair&lt;T1,T2&gt; makePair(T1 t1, T2 t2) {...}</code></li> <li>◊ <b>Parametrisierung</b> der Methode (&lt;T1,T2&gt;) steht vor dem Rückgabetyt</li> </ul> </li> <li>▷ Aufruf: <ul style="list-style-type: none"> <li>◊ <code>Pair&lt;A,B&gt; pair1 = x.makePair(new A(), new B());</code></li> <li>◊ Compiler erkennt selbst die Typen für die Methode</li> </ul> </li> <li>▷ Falls T1 z.B. schon die Klasse X parametrisiert: <pre>public class X &lt;T1&gt; {     public &lt;T2&gt; Pair&lt;T1,T2&gt; makePair(T1 t1, T2 t2) {...} }</pre> </li> </ul>
Typparameter	<ul style="list-style-type: none"> <li>▷ Alle Arten von Klassen und Arrays möglich</li> <li>▷ Auch parametrisierte Klassen sind als Typparameter möglich</li> <li>▷ Typparameter dürfen jedoch nicht vom primitiven Datentyp sein</li> <li>▷ Vererbung von Typparametern ist jedoch nicht übertragbar <ul style="list-style-type: none"> <li>◊ Bei bereits instanziierten Parametern sind keine Subklassen möglich</li> </ul> </li> <li>▷ Kurzform: <ul style="list-style-type: none"> <li>◊ <code>Pair&lt;String,Integer&gt; pair;</code></li> <li>◊ <code>pair = new Pair&lt;&gt; ("Hello", 123);</code></li> <li>◊ "Diamond-Operator": Compiler erkennt selbstständig die Instanziierung</li> </ul> </li> </ul>
Eingeschränkte Typparameter	<ul style="list-style-type: none"> <li>▷ Werden bei der Definition von generischen Klassen/Methoden verwendet</li> <li>▷ <code>&lt;T extends X&gt; // T gleich X, oder direkt/indirekt Subtyp von X</code> <ul style="list-style-type: none"> <li>◊ Notwendig um sicherzustellen, dass aufgerufene Methoden definiert sind</li> <li>◊ z.B.: <code>&lt;T extends Number&gt; // z.B.: doubleValue() immer vorhanden</code></li> </ul> </li> <li>▷ Mehrfache Einschränkung: <ul style="list-style-type: none"> <li>◊ <code>&lt;T extends X &amp; Interface1 &amp; Interface2</code></li> <li>◊ Klasse muss, falls vorhanden, an erster Stelle stehen</li> </ul> </li> </ul>

Wildcards	<ul style="list-style-type: none"> <li>▷ Werden bei der Instanziierung von Typparametern verwendet</li> <li>▷ <code>public double m (X&lt;? extends Number&gt; n) {...}</code> <ul style="list-style-type: none"> <li>◊ Ermöglicht nun die Verwendung von Subklassen bei aktuellen Parametern</li> <li>◊ (Siehe Einschränkung Typparameter / 4. Stichpunkt)</li> </ul> </li> <li>▷ <b>Unterschied:</b> <ul style="list-style-type: none"> <li>◊ <code>public &lt;T extends Number&gt; double m (X&lt;T&gt; n) {...}</code></li> <li>◊ Generische Methode mit eingeschränkt wählbarem Typparameter</li> <li>◊ <code>public double m (X&lt;? extends Number&gt; n) {...}</code></li> <li>◊ Nichtgenerische Methode mit generischem Parameter mit eingeschränkt wählbarem Typparameter</li> </ul> </li> <li>▷ Weitere Wildcard: <code>X&lt;?&gt;</code> <ul style="list-style-type: none"> <li>◊ Allgemeinst möglichste, <code>extends Object</code></li> </ul> </li> <li>▷ <code>X&lt;? super Double&gt;</code> <ul style="list-style-type: none"> <li>◊ Mit allen Supertypen (direkt/indirekt) und alle implementieren Interfaces</li> </ul> </li> </ul>
Empfehlungen	<ul style="list-style-type: none"> <li>▷ Oracle-Empfehlungen im Bezug auf Wildcards</li> <li>▷ In-Parameter (Werte einer Methode, die nur gelesen werden): <ul style="list-style-type: none"> <li>◊ Verwendung von <code>extends</code></li> </ul> </li> <li>▷ Out-Parameter (Werte einer Methode, die nur geschrieben werden): <ul style="list-style-type: none"> <li>◊ Verwendung von <code>super</code></li> </ul> </li> <li>▷ In/Out-Parameter: <ul style="list-style-type: none"> <li>◊ Keine Verwendung von Wildcards</li> </ul> </li> <li>▷ Rückgaben: <ul style="list-style-type: none"> <li>◊ Keine Verwendung von Wildcards</li> </ul> </li> </ul>
Interface Comparator	<ul style="list-style-type: none"> <li>▷ <b>Functional Interface</b> im Package <code>java.util</code></li> <li>▷ Verwendung: <ul style="list-style-type: none"> <li>◊ Erstellen einer Vergleichsklasse, die <code>Comparator&lt;T&gt;</code> implementiert</li> <li>◊ <code>..class MyComp&lt;T extends Number&gt; implements Comparator&lt;T&gt; {...}</code></li> <li>◊ Generisch mit einem Typparameter</li> </ul> </li> <li>▷ Methode: <code>public int compare (T t1, T2) {...}</code> <ul style="list-style-type: none"> <li>◊ Methode, muss abhängig vom Fall, selbst implementiert werden</li> <li>◊ 0, falls beide Objekte äquivalent</li> <li>◊ Negative Zahl, falls 1.Objekt-Wert dem 2.Objekt-Wert vorangehend ist</li> <li>◊ Positive Zahl, falls 1.Objekt-Wert dem 2.Objekt-Wert nachfolgend ist</li> </ul> </li> <li>▷ <b>String</b> hat bereits eine Methode <code>compareTo</code>: sortiert lexikographisch</li> </ul>
Einschränkungen	<ul style="list-style-type: none"> <li>▷ Keine primitiven Datentypen als Instanziierung von Typparametern</li> <li>▷ Keine Erzeugung von Objekten/Arrays von Typparametern mit <code>new</code></li> <li>▷ Keine Klassenattribute von Typparametern</li> <li>▷ Kein Downcast oder <code>instanceof</code> von Typparametern</li> <li>▷ Kein <code>throw-catch</code> mit Typparametern</li> <li>▷ Keine Methodenüberladung mit Typparametern</li> </ul>

## 11 Graphics (java.awt.Graphics;)

Applet	<ul style="list-style-type: none"> <li>▷ leichtgewichtige Variante an Graphikprogrammen</li> <li>▷ <code>import java.awt.Applet;</code></li> <li>▷ 1. Erstellen eigener Applet-Klasse (<code>extends Applet</code>)</li> <li>▷ 2. Überschreiben der Methode <code>paint</code> <pre>public void paint (Graphics graphics) {...}</pre> <p>Klasse <code>Graphics</code> verknüpft Programm mit Zeichenfläche</p> </li> <li>▷ 2.1 <code>GeomShape2D</code>-Array <pre>GeomShape2D pic = new GeomShape2D[3];</pre> <p>Füllen des erstellten Arrays mit Formen (z.B.: <code>new Circle(0,0,0);</code>)</p> </li> <li>▷ 2.2 Erstellen jeder Form mithilfe Randfarbe, Füllfarbe und Zeichnen <pre>pic[0].setBoundaryColor(Color.RED); // Randfarbe pic[0].setFillColor(Color.RED); // Füllfarbe pic[0].paint(graphics); // Eigentliches Zeichnen</pre> </li> </ul>
GeomShape2D	<ul style="list-style-type: none"> <li>▷ Abstrakte Klasse (Methode <code>paint</code> ist abstrakt)</li> <li>▷ Attribute: <pre>int positionX; int positionY; int rotationAngle; int transparencyValue; Color boundaryColor; Color fillColor;</pre> </li> <li>▷ Subklassen: <code>Rectangle</code>, <code>Circle</code>, <code>StraightLine</code></li> </ul>

## 12 Interfaces

Erzeugung	<ul style="list-style-type: none"> <li>▷ Meist in eigener Datei</li> <li>▷ <code>public interface MyInterface {...}</code></li> <li>▷ Alle Methodes und das Interface <b>müssen public</b> sein</li> </ul>
Methoden	<ul style="list-style-type: none"> <li>▷ Objektmethoden werden hier nicht implementiert, sondern nur definiert</li> <li>▷ <code>public</code> kann weggelassen werden, da ohnehin notwendig</li> <li>▷ Implementierte Methoden müssen dann auch <code>public</code> sein</li> <li>▷ Falls eine der Methoden nicht implementiert wird ⇒ Klasse abstrakt</li> <li>▷ Klassenmethoden können definiert und implementiert werden <ul style="list-style-type: none"> <li>- Statischer Typ entscheidet bei Klassenmethoden welche Implementation</li> <li>- Deswegen ist dies hier in Java kein Problem</li> </ul> </li> <li>▷ Außerdem möglich: <ul style="list-style-type: none"> <li>◊ Klassenkonstanten (sind aber implizit <code>public</code> und <code>final</code>)</li> <li>◊ Implementierte "Default"-Objektmethoden</li> </ul> </li> </ul>
Default-Objektmethoden	<ul style="list-style-type: none"> <li>▷ Werden durch Schlüsselwort <code>default</code> vor Rückgabetyt eingeleitet</li> <li>▷ Problem bei Mehrfachvererbung von Interfaces: <ul style="list-style-type: none"> <li>◊ Compiler wirft Fehlermeldung, falls Klasse zwei Implementationen der selben Methode erbt</li> </ul> </li> <li>▷ Beispiel: <pre>1 public interface Int1 { 2     default void m() {...} 3 }</pre> </li> </ul>
Verwendung	<ul style="list-style-type: none"> <li>▷ <code>implements MyInterface</code> nach Klassenname</li> <li>▷ Beliebig viele Interfaces möglich (seperiert durch ,)</li> <li>▷ Ein Interface kann mehrere andere Interfaces erweitern (<code>extends</code></li> </ul>
Unterschiede Interface Abstrakte Klasse	<ul style="list-style-type: none"> <li>▷ Interfaces können Mehrfachvererbung</li> <li>▷ Abstrakte Klassen können von Klassen abgeleitet werden</li> <li>▷ Abstrakte Klassen, können non-<code>public</code> Attribute/Methoden haben</li> </ul>

## 13 JUnit-Tests

Allgemein	<ul style="list-style-type: none"> <li>▷ Tests als Ganzes - Black-Box-Tests</li> <li>▷ JUnit-Tests werden in eine separate Quelldatei geschrieben</li> <li>▷ Die zu testende Einheit/Klasse wird dann importiert</li> </ul>
Imports	<ul style="list-style-type: none"> <li>▷ <code>import static org.junit.Assert.assertEquals;</code></li> <li>▷ <code>import static org.junit.Assert.assertTrue;</code></li> <li>▷ <code>import org.junit.jupiter.api.Test;</code></li> <li>▷ <code>import org.junit.jupiter.api.BeforeEach;</code></li> <li>▷ <code>import static org.junit.jupiter.api.Assertions.assertThrows;</code></li> </ul>
Methoden:	<ul style="list-style-type: none"> <li>▷ <code>assertEquals(..., ...);</code> // true, falls beide Parameter identisch <ul style="list-style-type: none"> <li>◊ Existiert auch mit 3 Parametern, 3. Wert entspricht maximalen Unterschied</li> </ul> </li> <li>▷ <code>assertTrue(...);</code> // true, falls der Parameter true ist</li> <li>▷ <code>assertThrows(..., ...);</code> // Wirft Exception abhängig von Executable <ul style="list-style-type: none"> <li>◊ Erster Parameter zu werfende Exception.class</li> <li>◊ Zweiter Parameter Functional Interface aus dem Package java.lang.reflect</li> </ul> </li> </ul>
Test	<ul style="list-style-type: none"> <li>▷ <code>@Test</code> vor der Methode</li> <li>▷ <code>void</code> als Rückgabewert</li> <li>▷ Nutzung einer <code>assert</code>-Methode (siehe Methoden)</li> </ul>
BeforeEach	<ul style="list-style-type: none"> <li>▷ <code>@BeforeEach</code> vor der Methode</li> <li>▷ Wird vor jeder einzelnen Testmethode einmal ausgeführt</li> </ul>

## 14 Klassen

Erzeugung	<ul style="list-style-type: none"> <li>▷ meist in separater .java Datei</li> <li>▷ <code>public class MyClass {}</code></li> <li>▷ <code>new MyClass();</code> <ul style="list-style-type: none"> <li>◊ Reserviert ausreichend Speicherplatz für das Objekt</li> </ul> </li> <li>▷ <code>MyClass x = new MyClass();</code> <ul style="list-style-type: none"> <li>◊ Speichern der Adresse des neuen Objekts in der Referenz x</li> </ul> </li> </ul>
Attribute	<ul style="list-style-type: none"> <li>▷ Eigenschaften der Objekte/Klassen</li> <li>▷ z.B.: <code>private int x;</code> (Objektattribut)</li> <li>▷ z.B.: <code>private static int x;</code> (Klassenattribut)</li> </ul>
Konstruktor	<ul style="list-style-type: none"> <li>▷ Wird zur Erzeugung von neuen Objekten einer Klasse verwendet</li> <li>▷ Methode mit selben Namen wie Klasse und ohne Rückgabotyp</li> <li>▷ z.B.: <code>public MyClass (int x, int y) {this.x = x; this.y = y;}</code></li> <li>▷ Erzeugung eines neuen Objekts: <code>MyClass test = new MyClass(2,4);</code></li> <li>▷ Falls kein Konstruktor angegeben wird → Default Constructor <ul style="list-style-type: none"> <li>◊ Basisklasse muss auch Konstruktor mit leerer Parameterliste haben</li> </ul> </li> <li>▷ Konstruktoren werden <b>nicht</b> vererbt</li> <li>▷ <b>Static Initializer</b> <ul style="list-style-type: none"> <li>◊ Methodenkopf besteht nur aus <code>static {...}</code></li> <li>◊ Wird genutzt um auf jeden Fall Klassenkonstanten zu initialisieren</li> </ul> </li> <li>▷ Aufruf anderen Konstruktors in Konstruktor mit <code>this(Parameter);</code></li> </ul>
Abstraktion	<ul style="list-style-type: none"> <li>▷ <code>abstract public class MyClass {...}</code></li> <li>▷ Notwendig, sobald Klasse eine abstrakte Methode beinhaltet</li> <li>▷ Keine Objekterzeugung möglich</li> <li>▷ Meist als Klasse mit Rahmenbedingungen für Subklassen verwendet</li> </ul>
Klasse aller Klassen	<ul style="list-style-type: none"> <li>▷ <code>java.lang.Object</code></li> <li>▷ Jede Klasse ist direkt oder indirekt von <code>Object</code> abgeleitet</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>boolean equals (Object obj) {...}</code> // Test auf Wertgleichheit</li> <li>◊ <code>String toString() {...}</code> // Zustand des Objekts als String</li> <li>◊ Werden oft an jeweilige Klasse angepasst</li> </ul> </li> </ul>

Verborgene Informationen	<ul style="list-style-type: none"> <li>▷ Jedes Objekt einer Klasse erhält einen Verweis auf ein anonymes Objekt</li> <li>▷ Dieses anonyme Objekt wird für jede Klasse nur einmal eingerichtet</li> <li>▷ Enthält Informationen zur Klasse, Attribute und Methoden der Klasse</li> <li>▷ Methodentabelle: <ul style="list-style-type: none"> <li>◊ Gibt an, welche Implementationen aller Methoden verwendet wird</li> <li>◊ Ermöglicht, die Feststellung der Klasse zur Laufzeit</li> <li>◊ Methode in Supertyp und Subtyp haben den selben Index (Position)</li> </ul> </li> </ul>
Verschachtelte Klassen	<ul style="list-style-type: none"> <li>▷ Einbettung von Klasse in andere Klasse</li> <li>▷ Eingebettete Klasse sind ähnlich einem Attribut einer Klasse</li> <li>▷ Zum Beispiel: <pre> 1 public class X { 2     private class Y {...} 3 }</pre> </li> <li>▷ Y ist in diesem Fall die innere, X die äußere Klasse</li> <li>▷ Innere Klasse darf: <ul style="list-style-type: none"> <li>◊ Alle Identifier möglich</li> </ul> </li> <li>▷ Äußere Klasse darf: <ul style="list-style-type: none"> <li>◊ Nur <b>public</b> oder ohne <b>public</b>, kein <b>private</b> oder <b>protected</b></li> </ul> </li> <li>▷ Maximal eine Klasse darf <b>public</b> sein → Name der Quelldatei</li> <li>▷ Innere Klassen sind davon allerdings nicht betroffen</li> <li>▷ Objekterzeugung: <ul style="list-style-type: none"> <li>◊ Erstellung von Objekten der inneren Klasse über Objekt der äußeren Klasse</li> <li>◊ Automatische Erzeugung eines Verweises auf Erzeugungsobjekt</li> <li>◊ <b>X a = new X(); a.newY();</b></li> </ul> </li> <li>▷ Aufruf: <ul style="list-style-type: none"> <li>◊ <b>OuterClass.InnerClass x = ...;</b></li> <li>◊ Äußere Klasse + Innere Klasse durch Punkt getrennt</li> </ul> </li> <li>▷ <b>static</b>: <ul style="list-style-type: none"> <li>◊ <b>static</b> auch bei inneren Klassen möglich</li> <li>◊ Kann nur auf Klassenmethoden und -attribute zugreifen</li> <li>◊ Erzeugung ohne Objekt möglich z.B.: <b>X.Y a = new X.Y();</b></li> </ul> </li> </ul>

## 15 Konversionen

Implizit	<ul style="list-style-type: none"> <li>▷ Immer möglich, wenn kein Informationsverlust entstehen kann</li> <li>▷ z.B.: kleinerer Datentyp in größeren</li> </ul>
Explizit	<ul style="list-style-type: none"> <li>▷ Meist Informationsverlust</li> <li>▷ Durchführung durch Angabe des Datentyps in Klammern davor</li> <li>▷ z.B.: <b>int i = (int)testDouble;</b></li> </ul>

## 16 Methoden

Methodenaufbau	<ul style="list-style-type: none"> <li>▷ Modifier Rückgabewert Identifier (Parameterliste) {Anweisung}</li> <li>▷ Alles vor den Anweisung: Methodenkopf (Head)</li> <li>▷ Alles in den geschweiften Klammern: Methodenrumpf (Body)</li> <li>▷ z.B.: <code>public void setX (int x) {this.x = x;}</code> (Objektmethode)</li> <li>▷ z.B.: <code>public static void setY (int y) {this.y = y;}</code> (Klassenmethode)</li> <li>▷ <code>this.x</code> steht hier für das Objektattribut und nicht den Parameter</li> </ul>
Ausführung	<ul style="list-style-type: none"> <li>▷ Objektmethoden: <code>myObject.setX(2);</code></li> <li>▷ Klassenmethoden: <code>MyClass.setY(2);</code></li> </ul>
return	<ul style="list-style-type: none"> <li>▷ Wird für Rückgabe bei Methoden mit Rückgabewert benötigt</li> </ul>
Abstraktion	<ul style="list-style-type: none"> <li>▷ <b>abstract</b> vor Modifier (z.B.: <b>public</b>)</li> <li>▷ Abstrakte Methoden haben keinen Methodenrumpf</li> </ul>
Parameter	<ul style="list-style-type: none"> <li>▷ Parameterliste in Definition: Formale Parameter</li> <li>▷ Parameterliste bei Methodenaufruf: Aktuelle Parameter <ul style="list-style-type: none"> <li>◊ Kommt von actual ⇒ tatsächlich, vorliegend</li> </ul> </li> <li>▷ Verhalten bei Referenzen: <ul style="list-style-type: none"> <li>◊ Kopie der Adresse des Objekts bei Initialisierung des formalen durch aktuellen Parameter</li> </ul> </li> <li>▷ Variable Parameterzahl: <ul style="list-style-type: none"> <li>◊ <code>void m (double... args) {...}</code></li> <li>◊ Drei Punkte deuten variable Parameteranzahl an</li> <li>◊ Compiler macht aus den übergebenen Werten selbstständig ein Array</li> <li>◊ Ermöglicht variable Anzahl von Werten (<code>1.42,2.7</code>)</li> <li>◊ z.B.: Funktion, die das Maximum von übergebenen Variablen bestimmt</li> </ul> </li> </ul>
Signatur	<ul style="list-style-type: none"> <li>▷ Besteht aus Identifier und Parameterliste</li> <li>▷ Eine Klasse kann keine zwei Methoden mit derselben Signatur haben</li> </ul>
Klassenmethoden	<ul style="list-style-type: none"> <li>▷ Wird mithilfe von <b>static</b> zwischen Modifier und Rückgabewert definiert</li> <li>▷ Klassenmethoden werden über den Klassennamen aufgerufen</li> <li>▷ <b>Nicht erlaubt:</b> Lesen und Schreiben von Objektmethoden und -Attributen</li> <li>▷ <b>Nicht erlaubt:</b> Objektmethoden aufrufen</li> <li>▷ <b>Erlaubt:</b> Klassenattribute lesen und schreiben</li> <li>▷ <b>Erlaubt:</b> Klassenmethoden aufrufen</li> <li>▷ Workaround: Objekt als Parameter übergeben</li> <li>▷ <b>static</b>-Import funktioniert auch bei Klassenmethoden</li> <li>▷ Die Implementation wird hier durch den statischen Typ bestimmt</li> </ul>

## 17 Optional (java.lang.Optional;)

Informationen	<ul style="list-style-type: none"> <li>▷ Objekt der Klasse <b>Optional</b> kapselt ein Objekt seines Typparameters ein</li> <li>▷ Bietet bequemen Umgang mit der Möglichkeit, dass eine Referenz <b>null</b> ist</li> </ul>
Methoden	<ul style="list-style-type: none"> <li>◊ <b>ofNullable</b> <ul style="list-style-type: none"> <li>- Bekommt ein Objekt oder <b>null</b> übergeben und kapselt dieses ein</li> <li>- Gibt ein Objekt der Klasse <b>Optional</b> zurück</li> </ul> </li> <li>◊ <b>get</b> <ul style="list-style-type: none"> <li>- Liefert das eingekapselte Objekt zurück</li> <li>- Falls <b>null</b>: <b>NoSuchElementException</b></li> </ul> </li> <li>◊ <b>orElseGet</b> <ul style="list-style-type: none"> <li>- Zurücklieferung eines anderen Wertes vom Typparameter, falls <b>null</b></li> <li>- Formaler Parameter: <b>java.util.function.Supplier</b>;</li> </ul> </li> <li>◊ <b>ifPresent</b> <ul style="list-style-type: none"> <li>- Ausführung des Parameters, falls Objekt vorhanden (nicht <b>null</b>)</li> <li>- Formaler Parameter: <b>java.util.function.Consumer</b>;</li> <li>- z.B.: <b>opt1.ifPresent(x -&gt; {System.out.print(x);})</b>;</li> <li>- z.B.: Falls <b>opt1</b> ein Objekt einkapselt, wird es ausgegeben</li> </ul> </li> <li>◊ <b>map</b> <ul style="list-style-type: none"> <li>- Abbildung basierend auf Parameter</li> <li>- z.B.: <b>Optional&lt;Number&gt; opt2 = opt1.map(x -&gt; x * x)</b>;</li> <li>- z.B.: Hier <b>opt2</b> auch <b>null</b>, da <b>opt1 == null</b></li> </ul> </li> <li>◊ <b>filter</b> <ul style="list-style-type: none"> <li>- Liefert <b>Optional</b> vom selben generischen Typ zurück</li> <li>- Formaler Parameter: <b>java.util.function.Predicate</b>;</li> <li>- Filter <b>true</b>: Neues <b>Optional</b>-Objekt mit selbem Kapselinhalt</li> <li>- Filter <b>false</b>: Leeres <b>Optional</b>-Objekt wird zurückgegeben</li> <li>- z.B.: <b>Optional&lt;Number&gt; opt3 = opt1.filter(x -&gt; x + 2 == 1)</b>;</li> <li>- Gibt selbes Objekt zurück, falls Gleichung erfüllt</li> </ul> </li> </ul>
Beispiel	<ul style="list-style-type: none"> <li>▷ <b>Optional&lt;Number&gt; opt1 = Optional.ofNullable(null)</b>;</li> <li>▷ <b>Number n1 = opt1.get()</b>; // <b>NoSuchElementException</b></li> <li>▷ <b>Number n2 = opt1.orElseGet(() -&gt; 0)</b>; // Falls <b>null -&gt; 0</b></li> </ul>

## 18 Packages und Zugriffsrechte

Package	<ul style="list-style-type: none"> <li>▷ Zusammenfassung von mehreren Dateien</li> <li>▷ Wird zur Gruppierung von ähnlichen Funktionalitäten verwendet</li> <li>▷ Ermöglicht selbe Dateinamen in unterschiedlichen Packages</li> <li>▷ Bestehen nur aus Kleinbuchstaben</li> <li>▷ Am Anfang der Quelldatei: <b>package mypackage</b>;</li> <li>◊ Datei gehört damit zum Package <b>mypackage</b></li> <li>◊ <b>mypackage</b> wird automatisch importiert</li> </ul>
Import	<ul style="list-style-type: none"> <li>▷ <b>import package.*</b>;</li> <li>▷ <b>*</b> steht für alle Definitionen aus <b>package</b></li> <li>▷ <b>*</b> importiert aber nicht die Inhalte von Subpackages</li> <li>▷ Import-Anweisungen müssen immer am Anfang des Quelltextes stehen</li> <li>▷ Durch Importanweisungen sind Teile danach nur noch mit Namen ansprechbar</li> <li>▷ Wichtigstes Package: <b>java.lang.*</b> (automatisch importiert)</li> <li>▷ Konstanten: <b>import static java.lang.Math.PI</b>;</li> <li>◊ Ermöglicht Schreiben von <b>PI</b> statt <b>Math.PI</b></li> </ul>
Zugriffsrechte	<ul style="list-style-type: none"> <li>▷ Klassen/Enum: nur <b>public</b> oder nichts</li> <li>◊ Nur eine Klasse darf <b>public</b> sein (Damit auch Dateiname)</li> <li>▷ <b>private</b>: Zugriff innerhalb der Klasse</li> <li>▷ Keine Angabe: <b>private</b> + im Package</li> <li>▷ <b>protected</b>: Keine Angabe + in allen Subklassen</li> <li>▷ <b>public</b>: <b>protected</b> + an jeder Import-Stelle</li> </ul>

## 19 Programme und Prozesse

Quelltest	▷ z.B. selbst geschriebener Java-Code
Java-Bytecode	▷ Wird durch Übersetzung des Java-Quelltextes erzeugt
Programm	▷ Sequenz von Informationen
Aufruf eines Programms	▷ Starten eines Prozesses, der die Anweisungen des Programmes abarbeitet
Prozesse	▷ CPU besteht aus mehreren Prozessorkernen ▷ Mehrere Prozesse laufen dementsprechend parallel ▷ Allerdings bearbeitet jeder Kern nur einen Prozess gleichzeitig (sehr kurz) ◇ Illusion von Multitasking

## 20 Random (java.util.Random;)

Verwendung	▷ Erzeugung eines neuen Objekts ◇ <code>Random random = new Random();</code> ▷ Zahlenerzeugung mithilfe von: ◇ <code>random.nextInt();</code> ◇ <code>random.nextLong();</code> ◇ <code>random.nextFloat();</code> ◇ <code>random.nextDouble();</code> ▷ Bei float und double: Zwischen 0 und 0.1 ▷ Bei int und long: Zahl aus Wertebereich
Methoden	▷ <code>nextInt(), nextDouble(), ...</code> ◇ Generierung von Zufallszahlen ▷ <code>ints(), longs(), doubles()</code> ◇ Liefern jeweils Stream mit zufälligen Zahlen zurück ◇ In diesem Fall unendliche Länge ◇ Werden in Verbindung mit <code>IntStreams</code> (etc..) verwendet

## 21 Schleifen, if, switch

while-Schleife	▷ <code>while (Bedingung) {Anweisung;}</code> ▷ Schleife wird ausgeführt, solange die Bedingung wahr ist ▷ <code>{}</code> kann bei einzelner Anweisung auch weggelassen werden
do-while-Schleife	▷ <code>do {Anweisung;} while (Bedingung);</code> ▷ Anweisungsblock wird immer mindestens einmal ausgeführt
for-Schleife	▷ <code>for (Anweisung davor; Bedingung; Anweisung danach) {Anweisung}</code> ▷ z.B.: <code>for (int i = 0; i &lt; 10; i++) {...}</code> ◇ Zehnmalige Ausführung der Anweisung ▷ Kurzform: <code>for (Position p : positions) {}</code> ◇ (Komponententyp Identifier : ArrayName)
if-Anweisung	▷ <code>if (Bedingung) {...}</code> ◇ Führt den Code in der Anweisung nur aus, falls die Bedingung erfüllt ist ▷ <code>if (Bedingung) {} else {}</code> ◇ Code, der ausgeführt wird, falls Bedingung nicht erfüllt ist
switch-Anweisung	▷ Abfrage von mehreren Fällen ▷ <code>switch (i) { case 2: ... break; case 3: ... break; default: ... }</code> ▷ <code>break;</code> Ohne break, geht es mit der Anweisung für den nächsten Fall weiter ▷ Keine Variablen als Abfragen für Fälle / kein Ausdruck, nur EIN Wert ▷ <code>default</code> wird dann ausgeführt, wenn kein anderer Fall eintritt



## 22 Streams (java.util.stream.Stream;)

Information	<ul style="list-style-type: none"> <li>▷ Generisches Interface <b>Stream</b></li> <li>▷ Einheitliche Schnittstelle für Listen, Arrays, Dateien</li> <li>▷ Relevante Kapitel: <b>Optional</b></li> </ul>
Methodenzusammenfassung	<ul style="list-style-type: none"> <li>▷ filter, map, max, of</li> <li>▷ <b>filter</b> <ul style="list-style-type: none"> <li>◊ Liefert Stream vom selben generischen Typ zurück</li> <li>◊ Formaler Parameter: <code>java.util.function.Predicate</code>;</li> </ul> </li> <li>▷ <b>map</b> <ul style="list-style-type: none"> <li>◊ Liefert Stream von evtl. anderem Typparameter zurück</li> <li>◊ Dieser Typ ist abhängig vom aktuellen Parameter</li> <li>◊ Formaler Parameter: <code>java.util.function.Function</code>;</li> </ul> </li> <li>▷ <b>max</b> <ul style="list-style-type: none"> <li>◊ Liefert nur einzelnes Element zurück abhängig vom <b>Comparator</b></li> </ul> </li> <li>▷ <b>of</b> <ul style="list-style-type: none"> <li>◊ Dient der direkten Erzeugung von Streams</li> <li>◊ Beliebige Anzahl an Parametern des Typparameters</li> <li>◊ Rückgabe eines Streams mit diesen Elementen</li> <li>◊ z.B.: <code>Stream&lt;Number&gt;.of(new Integer(2), new Integer(3));</code></li> </ul> </li> <li>▷ <b>reduce</b> <ul style="list-style-type: none"> <li>◊ Erstellt aus allen Elementen des Streams ein einzelnes Ergebnis</li> <li>◊ Durch sukzessiven Aufruf der Funktion im aktuellen Parameter</li> <li>◊ z.B.: <code>String fileContent = stream.reduce(String::concat);</code></li> </ul> </li> </ul>
Stream aus Liste	<ul style="list-style-type: none"> <li>▷ <code>List&lt;Number&gt; list = new LinkedList&lt;Number&gt;();</code> // Erstellt Liste</li> <li>▷ <code>Stream&lt;Number&gt; stream1 = list.stream();</code> <ul style="list-style-type: none"> <li>◊ Liefert Stream vom selben generischen Typ</li> <li>◊ Methode der Klasse List</li> </ul> </li> <li>▷ ... <code>stream1.filter(myPred);</code> // Anwenden eines Filter</li> <li>▷ ... <code>stream1.map(myFct);</code> // Anwenden einer Abbildung</li> <li>▷ <code>Optional&lt;Number&gt; opt = stream.max(new MyComp());</code> <ul style="list-style-type: none"> <li>◊ Hier <b>Optional</b>, da der Stream auch leer sein kann</li> </ul> </li> <li>▷ Methoden wie <b>filter</b> und <b>map</b> werden <b>intermediate operations</b> genannt</li> <li>▷ Methoden wie <b>max</b> werden <b>terminal operations</b> genannt</li> <li>▷ Zusammenfassung dieser Operationen möglich:</li> <li>▷ ... <code>= list.stream().filter(myPred).map(myFct).max(new MyComp());</code></li> </ul>
Stream aus Array	<ul style="list-style-type: none"> <li>▷ <code>Number[] a = new Number[100];</code> // Erstellt Array</li> <li>▷ <code>Stream&lt;Number&gt; stream1 = Arrays.stream(a);</code> // Erzeugt Stream <ul style="list-style-type: none"> <li>◊ Aufruf der Arrays-Klassenmethoden <code>stream(Array a)</code></li> </ul> </li> </ul>
Iterator	<ul style="list-style-type: none"> <li>▷ <code>Iterator iter = stream.iterator();</code> // Erzeugt Iterator Objekt</li> <li>▷ <code>iter.hasNext()</code> // Verwendung als Abbruchbedingung</li> <li>▷ <code>iter.next()</code> // Zum Fortschreiten im Iterator</li> </ul>
Liste aus Stream	<ul style="list-style-type: none"> <li>▷ <code>List&lt;String&gt; list = stream.collect(Collectors.toList());</code> <ul style="list-style-type: none"> <li>◊ <b>Collectors</b> besitzt viele Klassenmethoden zur Verarbeitung von Streams</li> <li>◊ <code>toList()</code> liefert das generische Interface <b>Collector</b></li> </ul> </li> </ul>
Array aus Stream	<ul style="list-style-type: none"> <li>▷ <code>Number[] a = stream.toArray(Number[]::new);</code></li> <li>▷ Art der Erzeugung abhängig vom Parameter</li> <li>▷ Parameter: Siehe Methodennamen als Lambda-Ausdrücke</li> </ul>
Int-/Long-/DoubleStreams	<ul style="list-style-type: none"> <li>▷ Methoden sind genau diesselben wie bei normalen Streams</li> <li>▷ z.B.: <code>IntStream stream1 = IntStream.of(1,2,3);</code></li> <li>▷ Nutzen der Klasse <b>Random</b> für unendlichen Stream mit Zufallszahlen <ul style="list-style-type: none"> <li>◊ <code>IntStream stream1 = new Random().ints();</code></li> </ul> </li> </ul>

## 23 String (java.lang.String)

Eigenschaften	<ul style="list-style-type: none"> <li>▷ Sonderrolle, da Klasse, aber trotzdem Literale in Java</li> <li>▷ Zeichenketten, die aus allen möglichen chars bestehen</li> </ul>
Methoden:	<ul style="list-style-type: none"> <li>▷ <code>String str = "Hello World";</code> <ul style="list-style-type: none"> <li>◊ <code>str.length;</code> // 11</li> <li>◊ <code>str.charAt(2);</code> // e</li> <li>◊ <code>str.indexOf('e');</code> // 2</li> <li>◊ <code>str.matches("He.+rld");</code> // true</li> </ul> </li> <li>.+ ⇒ . als Platzhalter für beliebiges Zeichen, + erlaubt Wiederholung ⇒ Regular Expression</li> <li>◊ <code>String str 2 = str.concat("b");</code> // Anhängen</li> <li>◊ <code>String str 2 = str1 + "b";</code> // Kurzform</li> </ul>

## 24 Syntax

Keywords	<ul style="list-style-type: none"> <li>▷ Können nur an bestimmten Stellen im Code stehen</li> <li>▷ z.B. <code>class</code>, <code>import</code>, <code>public</code>, <code>while</code>,...</li> </ul>
Identifizier	<ul style="list-style-type: none"> <li>▷ Namen für Klassen, Variablen, Methoden,..</li> <li>▷ Erstes Zeichen darf keine Ziffer sein</li> <li>▷ Keine Keywords als Identifizier</li> <li>▷ Identifizier sind case-sensitive</li> </ul>
Konventionen	<ul style="list-style-type: none"> <li>▷ Variablen / Methoden beginnen mit Kleinbuchstaben (<code>testInt</code>)</li> <li>▷ Klassen beginnen mit Großbuchstaben (<code>testClass</code>)</li> <li>▷ Wortanfänge im Inneren mit Großbuchstaben</li> <li>▷ Konstanten bestehen aus <code>_</code> und Großbuchstaben (<code>CENTS_PER_EURO</code>)</li> <li>▷ Paketnamen nur aus Kleinbuchstaben und <code>_</code> bei unzulässigen Zeichen</li> <li>▷ Boolesche Bestandteile: Prädikate (<code>isGreen</code>)</li> <li>▷ Andere Bestandteile mit Wert: Beschreibung des Wertes</li> <li>▷ Subroutinen ohne Rückgabe: Imperativ (<code>fillOval</code>)</li> </ul>
Kommentare	<ul style="list-style-type: none"> <li>▷ <code>//</code> Einzelne Zeile</li> <li>▷ <code>/*...*/</code> Mehrere Zeilen</li> <li>▷ <code>/**...*/</code> Erzeugung von Javadoc</li> </ul>
Javadoc	<ul style="list-style-type: none"> <li>▷ Erzeugung mithilfe von <code>/**</code> und Enter</li> <li>▷ Bei Methodenköpfen: <ul style="list-style-type: none"> <li>◊ <code>@param x the dividend</code></li> <li>◊ <code>@return x divided by x</code></li> <li>◊ <code>@throws class IndexOutOfBoundsException if c is not an int</code></li> </ul> </li> <li>▷ Bei Quelldateien: <ul style="list-style-type: none"> <li>◊ <code>@author</code></li> <li>◊ <code>@version</code></li> </ul> </li> </ul>
Rechtsausdrücke	<ul style="list-style-type: none"> <li>▷ Haben Typ und Wert</li> <li>▷ z.B.: <code>2*3+1</code></li> </ul>
Linksausdrücke	<ul style="list-style-type: none"> <li>▷ Verweisen auf Speicherstellen</li> <li>▷ z.B.: <code>int n</code></li> </ul>

## 25 Threads

Interface Runnable	<ul style="list-style-type: none"> <li>▷ Aus Package <code>java.lang</code></li> <li>▷ Enthält den Inhalt des parallel laufenden Prozesses</li> <li>▷ <b>Functional Interface</b> mit funktionaler Methode <code>run</code></li> <li>▷ Funktionsweise: <ul style="list-style-type: none"> <li>◊ Erstellung einer Klasse, die das Interface <code>Runnable</code> implementiert</li> <li>◊ Implementierung der funktionalen Methode <code>run</code> <ul style="list-style-type: none"> <li>- <code>public void run() {...}</code></li> </ul> </li> <li>◊ Erzeugung eines Objekts unserer Klasse <ul style="list-style-type: none"> <li>- z.B.: <code>Runnable runnable = new MyRunnable(...);</code></li> </ul> </li> <li>◊ Erzeugung eines <code>Thread</code>-Objekts mithilfe unseres <code>runnable</code> <ul style="list-style-type: none"> <li>- <code>new Thread(runnable).start();</code></li> </ul> </li> <li>◊ Der <code>Thread</code> wird dadurch auch gestartet</li> </ul> </li> </ul>
Klasse Thread	<ul style="list-style-type: none"> <li>▷ Aus Package <code>java.lang</code></li> <li>▷ <code>Thread</code> organisiert einen parallel laufenden Prozess</li> <li>▷ Methoden: <ul style="list-style-type: none"> <li>◊ <code>static currentThread</code> <ul style="list-style-type: none"> <li>- Keine Parameter</li> <li>- Liefert den <code>Thread</code> in dem die Methode aufgerufen wurde</li> </ul> </li> <li>◊ <code>dumpStack</code> <ul style="list-style-type: none"> <li>- Schreiben den CallStacks auf <code>System.err</code></li> </ul> </li> <li>◊ <code>static getAllStackTraces</code> <ul style="list-style-type: none"> <li>- Liefert die CallStacks aller <code>Threads</code> als Map</li> </ul> </li> <li>◊ <code>getId</code> <ul style="list-style-type: none"> <li>- Jeder <code>Thread</code> besitzt eine ID von Typ <code>long</code></li> <li>- Diese ID ist einmalig und bleibt gleich</li> </ul> </li> <li>◊ <code>getName</code> <ul style="list-style-type: none"> <li>- Abfrage des nicht einmaligen Namens</li> </ul> </li> <li>◊ <code>getPriority; setPriority</code> <ul style="list-style-type: none"> <li>- Jeder <code>Thread</code> besitzt eine Priorität</li> <li>- Anfangs gesetzt und dauernd beschränkt durch Klassenkonstanten</li> </ul> </li> <li>◊ <code>static sleep</code> <ul style="list-style-type: none"> <li>- Anhalten des <code>Threads</code> für übergebene Pause (<code>long</code>)</li> </ul> </li> <li>◊ <code>getState</code> <ul style="list-style-type: none"> <li>- Gibt den Status des <code>Threads</code> aus</li> </ul> </li> </ul> </li> </ul>
Threads und Streams	<ul style="list-style-type: none"> <li>▷ Verknüpfung zweier <code>Threads</code> mithilfe von <code>PipedInput(OutputStream)</code></li> <li>▷ z.B.: Ungefähre Vorhergehensweise: <ul style="list-style-type: none"> <li>◊ Erzeugung beider Streams: (Beachten von <code>try-catch</code>): <ul style="list-style-type: none"> <li>- <code>PipedOutputStream out = new PipedOutputStream();</code></li> <li>- <code>PipedInputStream in = new PipedInputStream(out);</code></li> </ul> </li> <li>◊ Implementieren einer schreibenden <code>Runnable</code>-Klasse <ul style="list-style-type: none"> <li>- z.B.: Schreiben von zufälligen Zahlen auf <code>out</code></li> </ul> </li> <li>◊ Erstellen des <code>Threads</code>: <ul style="list-style-type: none"> <li>- <code>Runnable runnable = new MyWriteRunnable(out);</code></li> <li>- <code>new Thread(runnable).start();</code></li> </ul> </li> <li>◊ Lesen mithilfe in der geschriebenen Daten <ul style="list-style-type: none"> <li>⇒ Zwei verbundene Streams, einer schreibt, der andere liest</li> </ul> </li> </ul> </li> </ul>
Interferierende Threads	<ul style="list-style-type: none"> <li>▷ Reihenfolge der Zugriffe, bei Zugriff auf die selbe Ressource, ungewiss</li> <li>▷ z.B.: Gleichzeitiges Schreiben auf <code>StdOut</code> // <code>Standard Out</code> → <code>System.out</code></li> </ul>

Thread terminieren	<ul style="list-style-type: none"> <li>▷ Beispiel: <ul style="list-style-type: none"> <li>◊ Einfügen einer Boolean-Variable in dazugehöriger <b>Runnable</b>-Klasse</li> <li>◊ Ausführung von <b>run()</b> solange diese <b>false</b> ist</li> <li>◊ Setzen der Variable auf <b>true</b>, wenn terminiert werden soll</li> </ul> </li> <li>▷ Sobald die Methode <b>run</b> beendet ist, terminiert der <b>Thread</b></li> <li>▷ Andere Umsetzung: <ul style="list-style-type: none"> <li>◊ Einfügen einer <b>terminate()</b>-Methode in die <b>Runnable</b>-Klasse</li> <li>◊ Diese setzt z.B. die oben implementierte Variable auf <b>true</b></li> <li>◊ Zugriff auf diese Methode über das erzeugte <b>Runnable</b>-Objekt</li> </ul> </li> </ul>
Gründe für Threads	<ul style="list-style-type: none"> <li>▷ Parallelisierung <ul style="list-style-type: none"> <li>◊ Aufteilung der Arbeitslast</li> <li>◊ Oft jedoch nicht schneller, sondern langsamer</li> </ul> </li> <li>▷ Abspaltung von eigenständigen Programmteilen <ul style="list-style-type: none"> <li>◊ Starten und Vergessen</li> </ul> </li> </ul>
Parallelisierung von Streams	<ul style="list-style-type: none"> <li>▷ Bereits implementiert, automatische, effiziente Aufteilung</li> <li>▷ Methode <b>parallelStream()</b> <ul style="list-style-type: none"> <li>◊ Kann, aber muss nicht, aufteilen</li> <li>◊ Liefert den selben Stream als Rückgabetyt zurück</li> <li>◊ bequeme Möglichkeit zur Verarbeitung großer Datenmengen</li> </ul> </li> </ul>

## 26 Vererbung

Zweck	▷ Weitergabe von allen Methoden und Attributen
Verwendung	▷ <code>public class MySubClass extends MyClass {}</code>
Konstruktor	▷ Aufruf des Konstruktors der Superklasse mithilfe von <code>super(Parameter);</code> ▷ Dieser Aufruf erfolgt im Konstruktor der Subklasse ▷ z.B.: <code>public MySubClass (int x) { super(x);&lt;v&gt;</code>
Overwrite	▷ Methoden in Subklassen können auch neu geschrieben werden <ul style="list-style-type: none"> <li>◊ Die Implementation der Superklasse wird sozusagen überschrieben</li> </ul> ▷ Selber Name und Parameterliste notwendig ▷ Signatur der Methoden muss identisch sein <ul style="list-style-type: none"> <li>◊ Die anderen Bestandteile können variieren:</li> <li>◊ Zugriffsrechte dürfen in abgeleiteter Klasse erweitert sein</li> <li>◊ <code>private</code> → <code>ε</code> → <code>protected</code> → <code>public</code></li> <li>◊ Bei Referenztypen Rückgabebetyp durch Subtyp ersetzbar</li> <li>◊ Exceptionklassen durch Subtypen ersetzbar</li> </ul> ▷ Aufruf der überschriebenen Methode mit <code>super.m();</code> ▷ Exceptions: <ul style="list-style-type: none"> <li>◊ Exception Klasse darf durch Subtyp ersetzt werden</li> </ul>
Overload	▷ Methoden mit selbem Bezeichner, aber unterschiedlicher Parameterliste ▷ Die Methode wird überladen ▷ Konstruktoren kann man auch überladen <ul style="list-style-type: none"> <li>◊ Für manche Werte werden dann Standardwerte gesetzt</li> <li>◊ Anderer Konstruktor auch in Konstruktor aufrufbar (<code>this(1);</code>)</li> </ul> ▷ Alle Methoden einer Klasse müssen unterschiedliche Signatur haben
Subtypen	▷ Abgeleitete Klassen / Interfaces ( <b>extends</b> ) ▷ Überall wo ein Referenztyp (Supertyp) erwartet wird: <ul style="list-style-type: none"> <li>◊ Verwendung eines Objekts eines Subtyps möglich               <ul style="list-style-type: none"> <li>in Zuweisung an Variable</li> <li>als Parameterwert</li> <li>als Rückgabewert</li> </ul> </li> </ul>
Statischer Typ	▷ Der Typ, mit dem Referenz definiert wird ▷ Statischer Typ unveränderlich mit Referenz verknüpft ⇒ statisch ▷ z.B.: <code>X a = new Y();</code> ⇒ <code>X</code> hier statischer Typ ▷ <b>Entscheidet</b> , auf welche Attribute/Methoden zugegriffen werden darf <ul style="list-style-type: none"> <li>◊ Müssen im statischen Typ vorhanden sein (definiert oder ererbt)</li> </ul>
Dynamischer Typ	▷ Der Typ des Objekts einer Referenz, auf das diese Referenz ▷ Muss gleich dem statischen Typ oder ein Subtyp des statischen Typs sein ▷ Kann sich beliebig häufig ändern ⇒ dynamisch ▷ z.B.: <code>X a = new Y();</code> ⇒ <code>Y</code> hier dynamischer Typ ▷ <b>Entscheidet</b> , welche Implementation der Methode aufgerufen wird
Downcast	▷ <code>if (y instanceof X) {...}</code> <ul style="list-style-type: none"> <li>◊ Gibt <code>true</code> zurück, falls <code>y</code> (Variable von Referenztyp) gleich dem Typen von <code>X</code> oder ein Subtyp von <code>X</code> ist</li> </ul> ▷ Downcast <ul style="list-style-type: none"> <li>◊ Vorherige Überprüfung mit <code>isinstanceof</code></li> <li>◊ Ermöglicht z.B.: <code>X z;</code>  <code>z = (X) y;</code></li> <li>◊ <b>Warum?</b> Zugriff auf Funktionen, die nicht im statischen Typ existieren</li> </ul>
Garbage Collector	▷ Teil des Laufzeitsystems ▷ Wird selbstständig aufgerufen, um Objekte ohne Referenz zu löschen ▷ Kann zwecks Laufzeitoptimierung konfiguriert werden

## 27 Anhang: Interne Zahlendarstellung

### Ganze Zahlen

Ganzzahlige Datentypen	<ul style="list-style-type: none"> <li>▷ <b>byte</b>     8 Bits</li> <li>▷ <b>short</b>    16 Bits</li> <li>▷ <b>int</b>       32 Bits</li> <li>▷ <b>long</b>     64 Bits</li> </ul>
Binärdarstellung	<ul style="list-style-type: none"> <li>▷ <b>Nicht-negative</b> Zahlen: <ul style="list-style-type: none"> <li>◊ Führendes Bit = <b>null</b></li> </ul> </li> <li>▷ <b>Negative</b> Zahlen: <ul style="list-style-type: none"> <li>◊ Führendes Bit = <b>eins</b></li> </ul> </li> <li>▷ Führendes Bit auch <b>Vorzeichenbit</b> genannt</li> <li>▷ Größte darstellbare Zahl: <math>2^{N-1} - 1</math> // Jedes Bit außer dem Ersten gesetzt</li> </ul>
Umwandlung nicht-negativ in Dezimal	<ul style="list-style-type: none"> <li>▷ Veranschaulichung am Datentyp <b>byte</b> <ul style="list-style-type: none"> <li>◊ <b>byte</b> maximal 127, deswegen reichen Zehnerpotenzen bis 100</li> </ul> </li> <li>▷ Vorhergehensweise: (Beispiel 01101101 / 109) <ul style="list-style-type: none"> <li>◊ <math>01101101_2 / 01100100_2 = 00000001_2 \rightarrow "1?"</math> <ul style="list-style-type: none"> <li>- Ganzzahlige Division der Zahl durch 100 <math>\rightarrow</math> 1 Rest 9</li> </ul> </li> <li>◊ <math>01101101_2 \% 01100100_2 = 00001001_2</math> <ul style="list-style-type: none"> <li>- Rest der vorherigen Division <math>\rightarrow</math> 9</li> </ul> </li> <li>◊ <math>00001001_2 / 00001010_2 = 00000000_2 \rightarrow "10?"</math> <ul style="list-style-type: none"> <li>- Teilen durch 2.höchste Zehnerpotenz (<math>10^1</math>) <math>\rightarrow</math> 0</li> </ul> </li> <li>◊ <math>00001001_2 \% 00001010_2 = 00001010_2</math> <ul style="list-style-type: none"> <li>- Rest der vorherigen Division <math>\rightarrow</math> 9</li> </ul> </li> <li>◊ <math>00001001_2 / 00000001_2 = 00001001_2 \rightarrow "109"</math> <ul style="list-style-type: none"> <li>- Teilen durch kleinste Zehnerpotenz (<math>10^0</math>) <math>\rightarrow</math> 9</li> <li>- Teilen durch 1 natürlich überflüssig</li> </ul> </li> </ul> </li> </ul>
Unicode-Kodierung	<ul style="list-style-type: none"> <li>▷ Darstellung der Dezimalziffern: <ul style="list-style-type: none"> <li>◊ Setzen des 16er und 32er Bits auf 1</li> <li>◊ Also Addieren von 48</li> <li>◊ Bereich der Zahlenwerte: 48 – 57</li> </ul> </li> </ul>
Umwandlung Dezimal in Datentyp	<ul style="list-style-type: none"> <li>▷ Jede Zeichen der Zahl (z.B. 3856) stellt ja eine <b>char</b>-Zahl dar <ul style="list-style-type: none"> <li>◊ Subtrahieren von 48 (siehe Unicode Kodierung)</li> <li>◊ Multiplikation mit dazugehöriger Zehnerpotenz</li> <li>◊ Addieren der einzelnen Werte</li> </ul> </li> <li>▷ Schleife über Zehnerpotenzen, so oft wie die Zahl Ziffern enthält</li> </ul>
Negative Zahlen (Zweierkomplement)	<ul style="list-style-type: none"> <li>▷ Umwandlung von positiv nach negativ: <ul style="list-style-type: none"> <li>◊ Binärdarstellung der positiven Zahl</li> <li>◊ Umdrehen aller Bits (1-Komplement)</li> <li>◊ Addieren einer 1 (2-Komplement)</li> <li>◊ Auch rückwärts anwendbar</li> </ul> </li> <li>▷ Zweierkomplement ermöglicht einfache Darstellung der 0 (0000)</li> <li>▷ Zweierkomplement ermöglicht einfache Subtraktion: <ul style="list-style-type: none"> <li>◊ Setzen des Subtrahenden ins Zweierkomplement</li> <li>◊ Addieren der beide Werte</li> </ul> </li> <li>▷ Negativer Bereich ist um eins größer als Positiver</li> </ul>

Bitlogik	<ul style="list-style-type: none"> <li>▷ Überprüfung, ob Bit gesetzt ist: <pre> 1 public static boolean bitIsSet(int bitArray, int position) { 2     return bitArray &amp; (1 &lt;&lt; position) != 0; 3 } </pre> <ul style="list-style-type: none"> <li>◇ <b>bitArray</b>: binäre Informationsquelle, 32 Bits</li> <li>◇ <b>position</b>: auszulesende Stelle (31 MSB, 0 LSB)</li> <li>◇ <b>1 &lt;&lt; position</b>: Verschiebt Bitmuster um <b>position</b>-viele Stellen nach links <ul style="list-style-type: none"> <li>- Schiebt 1 damit an abzufragende Stelle</li> <li>- <b>Linksshift</b>-Operator füllt alles rechts mit Nullen auf</li> </ul> </li> <li>◇ <b>&amp;</b>: Bitweise <b>Verundung</b> (1, falls beide an der Stelle 1) <ul style="list-style-type: none"> <li>- Ergibt an abzufragender Stelle genau 1, wenn <b>bitArray</b> an der Stelle auch 1</li> <li>- Alle anderen Bits werden durch neues Bitmuster auf 1 gesetzt</li> </ul> </li> <li>◇ Am Ende Überprüfung mit <b>!= 0</b></li> </ul> </li> <li>▷ Setzen eines einzelnen Bits: <pre> 1 public static int setBit(int bitArray, int position) { 2     return bitArray   (1 &lt;&lt; position); 3 } </pre> <ul style="list-style-type: none"> <li>◇ Selbes Verfahren wie bei der Überprüfung eines Bits</li> <li>◇ Diesmal allerdings mit <b> </b>: Bitweise Veroderung <ul style="list-style-type: none"> <li>- Setzt das Bit an der gefragten Stelle immer auf 1</li> </ul> </li> </ul> </li> <li>▷ Nicht-Setzen eines einzelnen Bits: <pre> 1 public static int unsetBit(int bitArray, int position) { 2     return bitArray &amp; ~(1 &lt;&lt; position); 3 } </pre> <ul style="list-style-type: none"> <li>◇ Selbes Verfahren wie bei der Überprüfung eines Bits</li> <li>◇ Diesmal allerdings mit <b>~</b>: Komplement <ul style="list-style-type: none"> <li>- Setzt das Bit immer auf 0 aufgrund des Komplements und der Verundung</li> </ul> </li> </ul> </li> </ul>
----------	--

## Gebrochene Zahlen

Gebrochene Zahlen	<ul style="list-style-type: none"> <li>▷ <b>float</b>      32 Bits</li> <li>▷ <b>double</b>    64 Bits</li> </ul>
Probleme mit Ungenauigkeiten	<ul style="list-style-type: none"> <li>▷ Umkehrrechnungen liefern nicht genau den Ausgangswert</li> <li>▷ Untergehen kleinerer Zahl bei Addition extrem unterschiedlich großer Zahlen</li> <li>▷ Subtraktion fast gleich großer Zahlen führt mglw. zu inkorrekten Bits</li> <li>▷ Ersetzen des Tests auf Gleichheit durch "ausreichend nahe beieinander"</li> </ul>
Interne Darstellung	<ul style="list-style-type: none"> <li>▷ <b>+3.14159E17</b></li> <li>▷ <b>Vorzeichen</b>: Wird als binäre Information in einem einzelnen Bit abgespeichert</li> <li>▷ <b>Basis</b>: Im Literal zur <b>Basis 10</b>, intern zur <b>Basis 2</b></li> <li>▷ <b>Mantisse</b>: Die Gleitkommadarstellung der Zahl (3.14159)</li> <li>▷ <b>Exponent</b>: Hier 17, Angabe der zu multiplizierende Potenz</li> </ul>
IEEE 754	<ul style="list-style-type: none"> <li>▷ Standard Nr. 754 der Vereinigung von Elektrotechnikern und Informatiker <ul style="list-style-type: none"> <li>◇ Regelt die binäre Darstellung von Gleitkommazahlen</li> </ul> </li> <li>▷ <b>Vorzeichen</b>: 1 Bit, 1 bedeutet negativ</li> <li>▷ <b>Mantisse</b> und <b>Exponent</b> in normaler Binärdarstellung <ul style="list-style-type: none"> <li>◇ <b>float</b>: <ul style="list-style-type: none"> <li>- <b>Mantisse</b>: 23 Bits</li> <li>- <b>Exponent</b>: 8 Bits</li> </ul> </li> <li>◇ <b>double</b>: <ul style="list-style-type: none"> <li>- <b>Mantisse</b>: 52 Bits</li> <li>- <b>Exponent</b>: 11 Bits</li> </ul> </li> <li>◇ Ergibt mit dem einzelnen Bit für <b>Vorzeichen</b> die Bitanzahl</li> </ul> </li> <li>▷ <b>Unendlich</b> und <b>NaN</b>: <ul style="list-style-type: none"> <li>◇ Auftreten des Falls: <b>Exponent</b> besteht nur aus Einsen</li> <li>◇ <b>Mantisse</b> nur 0, dann <b>Unendlich</b> <ul style="list-style-type: none"> <li>- Trotzdem vorzeichenabhängig</li> </ul> </li> <li>◇ Sonst als <b>NaN</b> (Not a Number)</li> </ul> </li> </ul>

## 28 Anhang: Korrekte Software

### Korrektheit auf einzelnen Abstraktionsebenen

Lexikalische Ebene	<ul style="list-style-type: none"><li>▷ Darstellung typischer Fehler im Folgenden</li><li>▷ Rechtschreibung</li><li>▷ Formalisierung von Regeln:<ul style="list-style-type: none"><li>◇ Ähnliche Funktionsweise wie Grammatiken</li><li><code>identifizier ::= «letter» «ident-char-list»</code></li><li><code>ident-char-list ::= ε   «ident-char» «ident-char-list»</code></li><li><code>ident-char ::= «letter»   «digit»   _   \$</code></li><li><code>letter ::= a...z   A...Z</code></li><li><code>digit ::= 0...9</code></li><li>◇ <code>::=</code> Formale Definition von Sprachkonstrukten</li><li>◇ <b>Definiendum</b> links von <code>::=</code> (Name des Konstrukts)</li><li>◇ <b>Definiens</b> rechts von <code>::=</code> (Definierende Ausdruck)</li><li>◇ Verwendung von <code>«...»</code> bei Verwendung eines Konstrukts bei Definition</li><li>◇ <code> </code>: Trennt verschiedene Alternativen</li><li>◇ <code>ε</code> steht für das leere Wort</li><li>◇ Ableiten von korrekten <b>Identifizieren</b> mithilfe dieser Grammatik</li></ul></li></ul>
Syntaktische Ebene	<ul style="list-style-type: none"><li>▷ Definition Syntax:<ul style="list-style-type: none"><li>◇ Determiniert, ob ein Quelltext korrekt ist</li><li>◇ Vorgegebene Regeln</li><li>◇ Einfassen der kontextfreien Teile der Syntax in Regeln</li><li>◇ d.h. Ignorieren aller Zusammenhänge des Quelltextes<ul style="list-style-type: none"><li>- z.B. ob Variable typgerecht verwendet wird</li></ul></li></ul></li><li>▷ Syntaxfehler werden meist durch <b>Compiler</b> gefunden</li><li>▷ Korrekte Kammersetzung:<ul style="list-style-type: none"><li>◇ Zu jeder öffnenden Klammer genau eine nachfolgende schließende Klammer</li><li>◇ Zwei Klammerpaare immer nacheinander oder ineinander</li></ul></li><li>▷ Syntaktische Konstrukte:<ul style="list-style-type: none"><li>◇ Bildung anhand vorgegebener Struktur</li><li>◇ z.B. <b>for-Schleife</b>: <code>for(..;..;..)</code></li><li><code>statement ::= «compound-statement»   «if-statement»   ...</code></li><li><code>compound-statement ::= { «statement-sequence» }</code></li><li><code>statement-sequence ::= ε   «statement» «statement-sequence»</code></li><li>◇ Formale Definitionen erlauben Klärung von Detailfragen ("Darf ... leer sein?")</li><li><code>if-statement ::= if(«condition») «statement»  </code> <code>if(«condition») «statement» else «statement»</code></li><li>◇ Allgemein nützlich um die Syntax von Java nachzuvollziehen</li></ul></li></ul>
Semantische Ebene	<ul style="list-style-type: none"><li>▷ Definition Semantik:<ul style="list-style-type: none"><li>◇ Tatsächlicher Effekt eines sprach korrekten Programms</li><li>◇ Werden zur Laufzeit des Programms gefunden</li><li>◇ Werfen einer <b>RuntimeException</b></li></ul></li><li>▷ Beispiele:<ul style="list-style-type: none"><li>◇ Teilen durch 0</li><li>◇ Falscher <b>Array-Index</b></li><li>◇ Zugriff auf <b>null</b></li></ul></li></ul>
Logische Ebene	<ul style="list-style-type: none"><li>▷ Umsetzungsfehler, Denkfehler</li><li>▷ Fehler bei der Übertragung von eigentlich richtigen Gedanken</li><li>▷ Beispiel: <b>off-by-one error</b><ul style="list-style-type: none"><li>◇ Richtige Berechnung, aber um 1 "daneben"</li></ul></li><li>▷ Beispiel: Wochentag zu lang<ul style="list-style-type: none"><li>◇ z.B.: Reservierung von acht Zeichen</li><li>◇ <b>Wednesday</b> jedoch neun Zeichen lang</li></ul></li><li>▷ Logikfehler sind oft schwer zu finden</li></ul>



Spezifikatorische Ebene	▷ Spezifikatorischer Fehler: Bereits der umzusetzende Gedanke war falsch ▷ Beispiel: Jahr 2000 Problem ◇ Nicht gedacht, dass Programme bis Jahr 2000 im Dienst sind
-------------------------	---

## Korrektheit von Software

Korrektheit von Software	▷ Kein Programmabbruch durch Fehler ▷ Termination, wenn: <ul style="list-style-type: none"> <li>◇ Aufgabe erledigt</li> <li>◇ Befehl zur Termination von außen</li> </ul> ▷ Korrekte Ausgaben und Effekte
Korrektheit von Klassen	▷ Aufteilung in zwei Sammlungen von Aussagen: <ul style="list-style-type: none"> <li>◇ <b>Darstellungsinvariante</b> von Klassen und Interfaces             <ul style="list-style-type: none"> <li>- <b>representation invariant</b></li> <li>- Beschreibt die Darstellung der Objekte gegenüber dem Nutzer der Klasse</li> <li>- Die Sicht, die Attribute und Methoden vermitteln, die <b>public</b> sind</li> </ul> </li> <li>◇ <b>Implementationsinvariante</b> von Klassen             <ul style="list-style-type: none"> <li>- <b>implementation invariant</b></li> <li>- Analog zur Darstellungsinvariante</li> <li>- Behandelt den Teil der Klassendefinition, der nicht <b>public</b> ist</li> <li>- z.B.: Java-Kommentar in der Klassen-Quelldatei</li> </ul> </li> </ul> ▷ Umsetzungsbeispiele: <ul style="list-style-type: none"> <li>◇ Attribute <b>private</b> halten</li> <li>◇ Zugriff auf Attribute nur über Methoden gewähren</li> <li>◇ Überschreibung der geerbten Methode <b>clone()</b> und <b>equals</b> <ul style="list-style-type: none"> <li>- Falls <b>equals</b> überschrieben wird, sollte auch <b>hashCode</b> überschrieben werden</li> <li>- Anforderungen an <b>equals</b> zu finden in Dokumentation <b>java.lang.Object</b></li> </ul> </li> </ul> ▷ Formulierung Darstellungsinvariante Beispiel: Ein Objekt von Klasse <b>DMatrix</b> repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von <b>double</b> ; Zeilen- und Spaltenzahl sind konstant. <ul style="list-style-type: none"> <li>- Beschreibung aller Begrenzungen, Umsetzungen, etc</li> </ul> ▷ Formulierung Implementationsinvariante Beispiel: Attribut <b>matrix</b> vom Typ <b>double[] []</b> hat als Länge die Seitenzahl und seine Komponenten haben als Länge die Spaltenzahl. <b>matrix[i][j]</b> ist der Eintrag in Zeile <b>i</b> und Spalte <b>j</b> . <ul style="list-style-type: none"> <li>- <b>ALLE private</b>-Attribute sollten hier angesprochen werden</li> <li>- Falls nicht, sind sie auch nicht wichtig genug überhaupt zu existieren</li> <li>- Parallele Entwicklung der Implementationsinvariante und dem Projekt</li> </ul> ▷ Ableitung von Klassen / Implementationen von Interfaces <ul style="list-style-type: none"> <li>◇ Subtypen müssen Darstellungsinvariante einhalten             <ul style="list-style-type: none"> <li>- z.B. Methode in Subtyp muss selben Effekt auf Darstellungsinvariante haben</li> <li>- Liskov Substitution Principle</li> </ul> </li> <li>◇ Implementationsinvariante muss bei <b>protected</b>-Attributen der Basisklasse übernommen werden             <ul style="list-style-type: none"> <li>- <b>private</b>-Attribute nicht relevant, unter Kontrolle der Basisklasse</li> </ul> </li> <li>◇ Übernommen werden heißt:             <ul style="list-style-type: none"> <li>- Darf erweitert und verfeinert werden</li> <li>- <b>Nichts</b> darf zurückgenommen werden</li> </ul> </li> </ul>

Korrektheit von Subroutinen	<ul style="list-style-type: none"> <li>▷ Subroutine als Oberbegriff für Methoden/Funktionen</li> <li>▷ Vertrag zwischen dem Nutzer und dem Entwickler einer Subroutine <ul style="list-style-type: none"> <li>◊ Wenn der Aufruf alle Vorbedingungen erfüllt, muss die Subroutine alle Nachbedingungen erfüllen</li> <li>◊ Vorbedingungen: <ul style="list-style-type: none"> <li>- Implementationsinvariante vor dem Aufruf eingehalten</li> <li>- Parameter müssen gewisse Bedingungen erfüllen</li> <li>- Variable/Konstante außerhalb der Klasse</li> <li>- Externe Datenquellen (z.B. Dateien)</li> </ul> </li> <li>◊ Nachbedingungen: <ul style="list-style-type: none"> <li>- Implementationsinvariante nach dem Aufruf eingehalten</li> <li>- Rückgabewert muss von bestimmtem Typ sein</li> <li>- Variable außerhalb der Klasse</li> <li>- Externe Datenquellen (z.B. Dateien)</li> </ul> </li> </ul> </li> <li>▷ Aufbau des Vertrags: <ul style="list-style-type: none"> <li>◊ Type</li> <li>◊ Precondition</li> <li>◊ Returns</li> <li>◊ Postcondition</li> </ul> </li> <li>▷ Ableitung von Basisklasse / Implementationen von Interface: <ul style="list-style-type: none"> <li>◊ Vorbedingung darf nur abgeschwächt werden, nicht verschärft oder ersetzt</li> <li>◊ Nachbedingung darf nur verschärft werden, nicht abgeschwächt oder ersetzt</li> <li>◊ Zweiter Teil des Liskov Substitution Principle</li> </ul> </li> </ul>
Korrektheit von rekursiven Subroutinen	<ul style="list-style-type: none"> <li>▷ Rekursionsabbruch <ul style="list-style-type: none"> <li>◊ Muss vorhanden sein, damit Rekursion ordentlich terminiert</li> </ul> </li> <li>▷ Rekursionsschritt <ul style="list-style-type: none"> <li>◊ Schritt näher an den Rekursionsabbruch</li> </ul> </li> <li>▷ Beweis der Korrektheit mithilfe von Induktion: <ul style="list-style-type: none"> <li>◊ Induktionsbehauptung: Aufstellen für <b>Problemgröße</b></li> <li>◊ Induktionsanfang: z.B.: <b>Problemgröße</b> = 1</li> <li>◊ Induktionsvoraussetzung: Der Vertrag gelte für ...</li> <li>◊ Induktionsschritt: z.B.: Verringerung der Listenlänge</li> </ul> </li> </ul>
Korrektheit von Schleifen	<ul style="list-style-type: none"> <li>▷ Schleifeninvariante : Aussagen darüber, was sich während Schleife nicht ändert <ul style="list-style-type: none"> <li>◊ Formulierung: "Nach <math>h \geq 0</math> Schritten ist ... "</li> <li>- Verwendung einer Variable (h), die nicht im Code vorkommt</li> </ul> </li> <li>▷ Schleifenvariante : Aussagen darüber, was sich während Schleife ändert <ul style="list-style-type: none"> <li>◊ Formulierung: <b>for</b>: "h steigt um 1 "</li> <li>▷ Zusammenfassung: <ul style="list-style-type: none"> <li>◊ Formulierung: "Nach Schleifenende ist... "</li> </ul> </li> </ul> </li> <li>▷ Induktion bei Schleifen: <ul style="list-style-type: none"> <li>◊ Invariante = Induktionsbehauptung: "Nach <math>h \geq 0</math> Schleifendurchläufen gilt..."</li> <li>◊ Induktionsanfang, also <math>h=0</math>: "Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Durchlauf erfüllt ist."</li> <li>◊ Induktionsvoraussetzung für <math>h &gt; 0</math>: "Die Invariante gelte für <math>h-1</math>."</li> <li>◊ Induktionsschritt: "Unter Voraussetzung, dass..., nach h Durchläufen gilt."</li> </ul> </li> </ul>

## 29 Anhang: Effizienz von Software

### Nebenaspekte der Effizienz

Speicherplatz	<ul style="list-style-type: none"><li>▷ Heutzutage z.B. auch bei größeren Datenmengen noch relevant</li><li>▷ Begriffe:<ul style="list-style-type: none"><li>◊ Begrenzter Objektspeicher: <b>Heap</b></li><li>◊ Voller Speicher: <b>OutOfMemoryError</b></li></ul></li><li>▷ Regeln:<ul style="list-style-type: none"><li>◊ Vermeidung des unnötigen Festhaltens von Speicher</li></ul></li><li>▷ Memory Leaks in <b>C</b> und <b>C++</b><ul style="list-style-type: none"><li>◊ Manuelle Freigabe von Speicherbereichen auf dem <b>Heap</b> notwendig</li><li>◊ Führt bei Nichtbeachtung zu Memory Leaks</li></ul></li><li>▷ Speicherproblem bei Rekursion:<ul style="list-style-type: none"><li>◊ Nur ein begrenzter Bereich für <b>Call-Stack</b> reserviert</li><li>◊ Zu tiefe Rekursion → <b>StackOverflowError</b></li></ul></li></ul>
Netzwerkbelastung	<ul style="list-style-type: none"><li>▷ Wird in späteren Veranstaltungen noch intensiver aufgegriffen</li><li>▷ Generisches Beispiel: Dezentrale Datenhaltung<ul style="list-style-type: none"><li>◊ Daten werden redundant auf mehreren Netzknoten gehalten<ul style="list-style-type: none"><li>- Mindestens ein Server, auf den Nutzer Zugriff hat, vorhanden</li></ul></li><li>◊ Vorteile:<ul style="list-style-type: none"><li>- Je nach Belastungssituation, Weiterleitung zu anderem Server</li><li>- Keine langen Wege im Netzwerk</li></ul></li><li>◊ Nachteil:<ul style="list-style-type: none"><li>- Abgleich der Daten nach Änderung erfordert baldige Kommunikation</li></ul></li><li>◊ Geeignete Wahl der Serverstruktur vonnöten</li></ul></li></ul>
Reservierung von Ressourcen	<ul style="list-style-type: none"><li>▷ Ressourcen: Entitäten, die exklusiv von einem Prozess reserviert werden</li><li>▷ Wird später in Veranstaltungen zu Datenbanksystem näher behandelt</li><li>▷ z.B. Verwendung von <b>try-with-resources</b><ul style="list-style-type: none"><li>◊ Ressourcen werden automatisch in jedem Fall wieder geschlossen</li><li>◊ Ressourcen werden auf das zeitliche Minimum beschränkt</li></ul></li></ul>
Pareto Regel	<ul style="list-style-type: none"><li>▷ Allgemein statistische Regel aus der VWL</li><li>▷ Übertragung auf das Thema effiziente Software:<ul style="list-style-type: none"><li>◊ Nur wenige Quelltext ist für den Großteil der Ineffizienz verantwortlich</li></ul></li><li>▷ Konsequenz für Effizienzverbesserungen:<ul style="list-style-type: none"><li>◊ 1. Prüfen wo die Effizienzverluste auftreten</li><li>◊ 2. Bei Verbesserungen auf diese Stellen konzentrieren</li></ul></li></ul>

## Hauptaspekt der Effizienz - Laufzeit

Laufzeit messen

- ▷ Grundlegende Unterscheidungen:
  - ◊ Gewöhnliche Zeit vs CPU-Zeit
    - Gewöhnliche Zeit: wieviel Zeit seit dem Start vergangen ist
    - CPU-Zeit: Wieviel Rechenzeit der Thread bisher hatte
    - CPU-Zeit deswegen meist effizientere Betrachtung
  - ◊ User Time vs System Time
    - User Time: bislang verbrauchte CPU-Zeit für Prozess
    - System Time: Vom System für den Prozess verbrauchte CPU-Zeit
    - CPU-Zeit = User Time + System Time
- ▷ Gewöhnliche Zeit Messung:

```
1 long startTime = System.currentTimeMillis();
2 ..some code..;
3 System.out.print(System.currentTimeMillis() - startTime);
```

  - ◊ auch noch Methode `nanoTime()`
    - Jedoch nur Garantie, dass diese auf Millisekunden genau ist
- ▷ CPU-Zeit Messung:
  - ◊ Messung im aufgerufenen Thread:

```
1 import java.lang.management.*;
2 ThreadMXBean bean = ManagementFactory.getThreadMXBean();
3 long startTimeCpu = bean.getCurrentThreadCpuTime();
4 long startTimeUser = bean.getCurrentThreadUserTime();
5 long cpuTime = bean.getCurrentThreadCpuTime() - startTimeCpu;
6 long userTime = bean.getCurrentThreadUserTime() - startTimeUser;
7 long systemTime = cpuTime - userTime;
```
  - ◊ Messung in anderen Threads:

```
1 long totalTime = 0;
2 for (Thread thread : threads) {
3     long time = bean.getThreadCpuTime(thread.getId());
4     if (time != -1) totalTime += time;
5 }
```

    - Zeile 2: Durchlauf durch alle Threads
    - Zeile 3: Abfrage der verbrauchten CPU-Zeit eines Threads
    - Zeile 4: Falls Thread schon terminiert, wird -1 zurückgeliefert
- ▷ Kommerzielle Tools zur Zeitmessung (Profiler):
  - ◊ JVM Profiler:
    - CPU-Zeit, Methodenaufrufe, Speichernutzung
    - Abgriff dieser Daten direkt an der JVM
  - ◊ Instrumentalisierende Profiler:
    - Fügen weiteren Code zum Monitoring ein
    - "instrumentalisieren" den zu überwachenden Code
    - Jedoch zusätzliche Laufzeit
  - ◊ Application Performance Monitoring (APM):
    - instrumentalisierende Profile mit minimalistischer Datensammlung
    - Geringer Laufzeit-Overhead
    - Gesammelten Daten geben jedoch oft nur Hinweise
    - Geeignet für Monitoring im produktiven Einsatz

#### Laufzeitverbesserungen

- ▷ Assert-Anweisungen abschalten
- ▷ Werte nicht mehrfach berechnen
  - ◊ z.B. Verwendung einer Methodenrückgabe für **for**-Schleifenbedingung
  - Stattdessen Abspeichern der Rückgabe in Konstante
  - ◊ Verwendung von Konstanten bei Doppeltberechnungen
  - Aufpassen auf Seiteneffekte, die z.B. durch mehrfachen Aufruf entstehen
- ▷ Primitive Datentypen sind schneller
  - ◊ Verwendung statt Wrapper-Klassen
  - ◊ Verzicht auf Generizität an gewissen Stellen
  - ◊ z.B. Konversion von Datenstrukturen für Berechnung in Effizientere
  - ◊ Verzicht auf **BigInteger** und **BigDecimal**
  - deutlich höhere Laufzeit
- ▷ Inlining
  - ◊ Direkte Angabe eines Wertes statt der Benutzung der **get()**-Methode
- ▷ Unrolling bei Schleifen
  - ◊ Falls Fortsetzungsbedingung aufwendiger als Anweisungsblock ist
  - ◊ Ausführen der Anweisung mehrmals in einem Durchlauf
- ▷ **StringBuilder** bzw **StringBuffer** statt +
  - ◊ Durch + werden immer neue String Objekte erzeugt
  - ◊ **StringBuilder** um Strings ohne neue Objekte aufzubauen
  - **StringBuilder str = new StringBuilder("Hello");**
  - **str.append("!");**
  - **str.insert(5, "World");**
  - ◊ **StringBuffer** etwas langsamer, aber mehrere Threads möglich
- ▷ Speicherplatz spendieren
  - ◊ Opfern von Speicherplatz für bessere Laufzeit
- ▷ Cache-Awareness
  - ◊ Daten werden aus dem Cache über Register in die CPU geladen
  - ◊ Nach Verarbeitung über Register wieder in den Cache
  - ◊ Immer feste Größe an Bytes zwischen Cache und Hauptspeicher transferiert
  - ◊ Zugriffe auf Daten, die nicht im Cache sind: **Cache Misses**
  - **Cache Misses** kosten viel vergleichsweise viel Laufzeit
  - ◊ z.B.: Durchlauf eines Matrix-Arrays in sinnvoller Reihenfolge
- ▷ Minimierung Anzahl Zugriffe auf Hauptspeicher
  - ◊ Hauptspeicher: z.B. Festplatten
  - ◊ Auch Kopie fester Größe in den Hauptspeicher: **Seite**
  - ◊ Zugriff auf Information, die nicht im Hauptspeicher ist: **Page Fault**
  - ◊ Datenstruktur **B-Baum**:
    - Minimiert Anzahl der Zugriffe auf Hauptspeicher
    - Wird in fast jedem Datenbanksystem verwendet
    - Näheres in AuD-Veranstaltung
- ▷ Threads vermeiden
  - ◊ Können Laufzeit verbessern, aber auch eventuell verschlechtern
  - ◊ Threads zur Designverbesserung fragwürdig
- ▷ Tools suchen und verwenden
  - ◊ Aggressive Optimierung:
    - Java Byte Code nicht besonders gut optimiert
    - **Virtual Dispatch** an vielen Stellen wegoptimieren
  - ◊ Native Code Compilation:
    - Übersetzung in Maschinencode statt Java Byte Code
    - Ist um Größenordnungen schneller, aber nicht portabel

## Asymptotische Komplexität

- ▷ Zur Vereinfachung: erst nur **User Time**
  - ◊ Schätzung der Laufzeit durch eine mathematische Funktion
    - In Kennzahlen, die die Problemgröße beschreiben
    - Mathematische Überlegungen und/oder empirische Laufzeitstudien
    - Es können auch mehrere Problemgrößen vorhanden sein
  - ◊ Representative Operation Counts
    - Identifikation der Anweisungen, die die Laufzeit dominieren könnten
    - Ausführungen zählen für mathematische Überlegungen
    - Laufzeiten akkumulieren bei Laufzeitstudien
    - Akkumulation: Verwendung der Zeitmessung + Akkumulator
- ▷ Asymptotische Komplexität am Beispiel der linearen/binären Suche:
  - ◊ Asymptotische Komplexität (AK) gibt an, in welcher Größenordnung die Laufzeit des Algorithmus mit der Problemgröße wächst
    - AK von linearer Suche ist linear
    - AK von binärer Suche ist logarithmisch
- ▷ **Worst Case** und **Best Case** am selben Beispiel
  - ◊ Problemgröße hier: Zahl  $N$  der zu durchsuchenden Werte
    - also die Länge des Arrays
  - ◊ Für jede Problemgröße gibt es einen **Worst** und **Best Case**
    - Zwei mathematische Funktionen in der Problemgröße
  - ◊ **Best Case**:
    - Fall, der die geringste Laufzeit produziert
    - Hier: das erste angeschautete Element ist größer/gleich dem gesuchten
    - Die Laufzeit im **Best Case** hängt **hier** nicht von  $N$  ab
  - ◊ **Worst Case**:
    - Fall, der die größte Laufzeit produziert
    - Hier: Man muss die ganze Schleife durchlaufen
    - Lineare Suche:  $N$  Durchläufe
    - Binäre Suche: ca.  $\log_2 N$  Durchläufe
  - ◊ Bei großen Werte von  $N$  alle Operationen außer Schleife unerheblich
  - ◊ Die Laufzeit pro Durchlauf variiert nur in engen Grenzen
  - ◊ Die Laufzeiten pro Durchlauf bewegen sich in relativ engen Korridor
    - $[c_1 * N \dots c_2 * N]$  bei linearer Suche im **Worst Case**
    - $[c_3 * \log_2 N \dots c_4 * \log_2 N]$  bei binärer Suche im **Worst Case**
  - ◊ Im **Best Case** bei beiden:  $[c_5 \dots c_6]$  **unabhängig von  $N$**
- ▷ Schreibweise
  - ◊ Seien  $f : \mathbb{N} \rightarrow \mathbb{R}$  und  $g : \mathbb{N} \rightarrow \mathbb{R}$
  - ◊ Annahme: Es gibt beliebige, aber feste  $c_u, c_o \in \mathbb{R}$  ( $0 < c_u \leq c_o$ ), so dass ab einer gewissen Größe der Eingabe  $n$  gilt:
 
$$c_u * g(n) \leq f(n) \leq c_o * g(n).$$
  - ◊ Dann schreiben wir:  $f \in \Theta(g)$ .
    - $\Theta$ : Menge aller Funktionen, die asymptotisch äquivalent zu  $g$  sind
    - Korridor um die eine Funktion, die von der anderen nicht verlassen wird: asymptotisch gleich
  - ◊ Bei einer konstanten Funktion  $g$  schreiben wir:  $f \in \Theta(1)$ 
    - Konstante Vergleichsfunktion,  $f$  bleibt in einem horizontalen Korridor
  - ◊ Laufzeit bei linearer/binärer Suche:
    - Lineare Suche im **Worst Case**:  $\in \Theta(N)$
    - Binäre Suche im **Worst Case**:  $\in \Theta(\log_2 N)$
    - Beide im **Best Case**:  $\in \Theta(1)$
- ▷ Grenzen der Asymptotik
  - ◊ Nicht sinnvoll, wenn es um kleine Problemgrößen geht
  - ◊ Sehr viele kleine Probleme können aber trotzdem zu Laufzeitproblemen führen

Untere und  
obere Schranken

- ▷ Asymptotisches Verhalten lässt sich oft nicht genau einschätzen
  - ◊ Verwendung von oberen und unteren Schranken
- ▷ Zwei mathematische Funktionen  $g_u$  und  $g_o$ , so dass  $f$ 
  - ◊ **mindestens** so schnell wie  $g_u$  und
  - ◊ **höchstens** so schnell wie  $g_o$  wächst.
- ▷ Bis jetzt als asymptotischen Vergleich nur  $\Theta()$  für asymptotische Gleichheit
- ▷ Schreibweise für größer/kleiner-Vergleich:
  - ◊ Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ .
  - ◊ Wir schreiben  $f \in o(g)$ , wenn  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ 
    - $g(n)$  wächst schneller als  $f(n)$
  - ◊  $f \in o(g)$  und  $f \in \Theta(g)$  schließen sich logisch aus
    - Gleiche Asymptotik und echt unterschiedliche Asymptotik schließen sich aus
    - Die schneller wachsende Funktion verlässt den Korridor um die Langsamere
  - ◊ Gibt auch Funktionen, wo weder  $f \in o(g)$  oder  $f \in \Theta(g)$  gilt
    - Funktionen sind unvergleichbar
- ▷ Kleiner-gleich/Größer-gleich:
  - ◊ Seien wieder  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ .
  - ◊  $f \in O(g)$ , wenn  $f \in \Theta(g)$  oder  $f \in o(g)$  ist
  - ◊  $f \in \Omega(g)$ , wenn  $g \in \Theta(g)$  oder  $g \in o(f)$  ist
  - ◊ Offensichtlich gilt  $f \in O(g)$  genau dann, wenn  $g \in \Omega(f)$  gilt
    - $f$  kleiner-gleich  $g \rightarrow g$  größer-gleich  $f$
- ▷ Regeln für  $\Theta, O, \Omega$  und  $o$  (Seien  $f, g, h : \mathbb{N} \rightarrow \mathbb{R}^+$ )
  - ◊  $\Theta$  induziert eine Äquivalenzrelation, das heißt, es gilt:
    - **Reflexiv:**  $f \in \Theta(f)$ 
      - Funktion verläuft in einem Korridor um sich selbst
    - **Symmetrisch:** Wenn  $f \in \Theta(g)$ , dann ist  $g \in \Theta(f)$ 
      - Gegenseitiges Verlaufen im jeweils anderen Korridor
    - **Transitiv:** Wenn  $f \in \Theta(g)$  und  $g \in \Theta(h)$ , dann auch  $f \in \Theta(h)$ 
      - Transitive Schlussfolgerung über Korridorverläufe
  - ◊  $O$  induziert eine partielle Ordnung bzgl.  $\Theta$ , das heißt, es gilt:
    - Reflexiv:  $f \in O(f)$
    - Antisymmetrisch: Wenn  $f \in O(g)$  und  $g \in O(f)$ , dann ist  $f \in \Theta(f)$
    - Transitiv: Wenn  $f \in O(g)$  und  $g \in O(h)$ , dann auch  $f \in O(h)$
  - ◊  $o$  induziert die strikte partielle Ordnung zu  $O$ :
    - Antireflexiv (d.h.  $f \notin O(f)$ ), antisymmetrisch und transitiv

## Average Case

- ▷ Ausgangssituation:
  - ◊ **Worst Case** und **Best Case** gehen ernsthaft auseinander
  - ◊ Der Algorithmus wird sehr viele Male aufgerufen
- ▷ Ziel:
  - ◊ durchschnittliche Laufzeit durch mathematische Funktion beschreiben
    - **Average Case**
- ▷ Methodisches Problem:
  - ◊ Basiert darauf, wie wahrscheinlich die möglichen Eingaben sind
  - ◊ Wahrscheinlichkeitsverteilung / Erwartungswert
    - Oft nicht einmal ungefähr bestimmbar
    - Daher **Average Case** nur selten theoretisch betrachtet
    - Jedoch Laufzeitstudien auf den realen Daten
- ▷ Beispiel Primzahltest:
  - ◊ 1. Alle Zahlen  $2 \dots N$  sind gleich wahrscheinlich
    - $\Omega(\sqrt{n} / \log_e n)$  und  $O(\sqrt{n})$  im **Average Case**
    - **Worst Case**  $O(\sqrt{n})$  ist obere Schranke für **Average Case**
  - ◊ 2. Primzahlen und Nichtprimzahlen sind gleich wahrscheinlich
    - $\Theta(\sqrt{n})$  im **Average Case**
  - ◊ 3. Nur gerade Zahlen
    - $\Theta(1)$  im **Average Case**
- ▷ Beispiel Lineare Suche:
  - ◊ Alle Suchwerte  $0 \dots \text{Integer.MAX\_VALUE}$  sind gleich wahrscheinlich
    - $\Theta(n)$  im **Average Case** bei normalen Werten im Array
  - ◊ Nur Zahlen, die im Array sind, werden gesucht
    - $\Theta(n)$  im **Average Case**
  - ◊ Nur Zahlen, die nicht größer, als der kleinste Wert im Array sind
    - $\Theta(1)$  im **Average Case**



## 30 Anhang: Fehlersuche und fehlervermeidender Entwurf

### Fehlersuche

Fehlersuche	<ul style="list-style-type: none"><li>▷ Auftreten des Fehlers vs Ursache des Fehlers<ul style="list-style-type: none"><li>◊ Stelle, an der Fehler auftritt nicht immer Stelle, an der Fehler passiert ist</li><li>◊ Zurückverfolgen der Fehlerursache</li></ul></li><li>▷ Fehlerursache:<ul style="list-style-type: none"><li>◊ Wo zum ersten Mal eine Vor-/Nachbedingung, In-/Variante nicht erfüllt ist</li></ul></li><li>▷ Finden des Fehlers mithilfe von <b>Assert</b>-Anweisungen</li><li>▷ Wenn Fehlerursache nicht klar:<ul style="list-style-type: none"><li>◊ Weitere <b>Assert</b>-Anweisungen, die weiter zurückgehen</li><li>◊ Temporäre <b>Assert</b>-Anweisungen für kritischen Datensatz<ul style="list-style-type: none"><li>- Hinzufügen eines bestimmten Kommentars für späteres Entfernen</li></ul></li></ul></li><li>▷ Fehler in Bibliothekskomponente eher auszuschließen</li><li>▷ Falls undurchschaubar: verdächtigen Quelltext neu implementieren</li></ul>
Laufzeittests	<ul style="list-style-type: none"><li>▷ Black-Box-Tests<ul style="list-style-type: none"><li>◊ Testen einer Klasse von außen, ohne hineinzuschauen</li><li>◊ Verwenden von <b>JUnit</b>-Tests</li><li>◊ Jedoch auch spezifischere <b>Assert</b>-Anweisungen möglich</li><li>◊ Prüfung der Darstellungsinvariante bei Klassen/Interfaces</li><li>◊ Prüfung der Nachbedingung bei Subroutinen</li></ul></li><li>▷ White-Box-Tests<ul style="list-style-type: none"><li>◊ Hauptsächlich <b>Assert</b>-Anweisungen</li><li>◊ Implementationsinvarianten von Klassen</li><li>◊ Vor- und Nachbedingungen von Anweisungen</li><li>◊ Schleifen(in)varianten</li></ul></li><li>▷ Vorhergehensweise:<ul style="list-style-type: none"><li>◊ Testumgebung parallel zum eigentlichen Code entwickeln</li><li>◊ Änderung der Testumgebung falls Code-Änderung</li><li>◊ Beim Finden eines Fehlers: Einbau eines neuen Tests</li></ul></li><li>▷ Coverage/Fehlerabdeckung:<ul style="list-style-type: none"><li>◊ Randfälle:<ul style="list-style-type: none"><li>- z.B.: Dreiecke: Alle Ecken auf einer Linie</li></ul></li><li>◊ Bei Verzweigungen: jeder mögliche Pfad:<ul style="list-style-type: none"><li>- Überprüfen jedes verschiedenen Ergebnisses</li></ul></li><li>◊ <math>n</math>-verknüpfte <b>if</b>-Abfragen = <math>2^n</math> Fälle zu testen</li></ul></li></ul>

### Fehlervermeidender Entwurf

Verbesserung	<ul style="list-style-type: none"><li>▷ Prinzipien und Techniken für fehlervermeidenden Entwurf verbessern auch:<ul style="list-style-type: none"><li>◊ Wartbarkeit</li><li>◊ Modifizierbarkeit</li><li>◊ Erweiterbarkeit</li></ul></li></ul>
KISS	<ul style="list-style-type: none"><li>▷ "keep it simple, stupid!"</li><li>▷ Dekomposition in kleine, überschaubare Einheiten (Klassen, Subroutinen, etc.)</li><li>▷ Programm muss nicht unbedingt besonders "raffiniert" sein</li><li>▷ Zerlegung in intuitiv sofort verständliche, realitätsabbildende Einheiten</li><li>▷ Gut gewählte Identifier für Einheiten</li><li>▷ Verwendung der für Java festgelegten Namenskonventionen<ul style="list-style-type: none"><li>◊ Generelle Regel:<ul style="list-style-type: none"><li>- <b>Alle wichtigen</b> Aspekte zu Wortbestandteilen machen</li><li>- Identifier ist der einzige Kommentar, der bei Nutzung dabei steht</li><li>- Falls Identifier zu lang werden → Struktur gemäß KISS überdenken</li></ul></li><li>◊ Ausnahmen:<ul style="list-style-type: none"><li>- Typische mathematische Notation (x,y,..)</li><li>- Packagenamen eher kurz</li><li>- Verwendung von allgemein bekannten Abkürzungen</li></ul></li></ul></li></ul>

Separation of  
Concerns (SoC)

- ▷ Zerlegung der Gesamtaufgabe in verschiedene Aspekte
- ▷ Beispiele in der Java-Standardbibliothek:
  - ◊ Runnable vs Thread
    - Zerlegung der Funktionalität von Threads in zwei Concerns
  - ◊ Component vs Listener vs Event
  - ◊ Collections.sort vs Comparator
- ▷ Sinn von SoC:
  - ◊ Übersichtlichere Programmstruktur
  - ◊ Wiederverwendbarkeit
  - ◊ Austauschbarkeit - selektiv und unabhängig

## Model-View-Controller

- ▷ Wichtigstes Beispiel für SoC - MVC
- ▷ Relevant für Programme mit starkem GUI-Bezug
- ▷ Logikteile des Programms von Sachen, wie der Darstellung, separieren
- ▷ Aufbau:
  - ◊ **Model:**
    - Die eigentliche Logik des Programms
  - ◊ **View:**
    - Darstellung der Modelldaten
    - Interaktion mit dem Nutzer
  - ◊ **Controller:**
    - Verwaltung des Programmlaufs
    - Häufig eher klein
- ▷ Beispiel Schachprogramm:
  - ◊ **Model:**
    - In welcher Zeile/Spalte Figur steht
    - Vorwissen über den Spieler
    - Subroutine zur Berechnung des nächsten Zugs
  - ◊ **View:**
    - Darstellung des Schachbretts auf dem Bildschirm
    - Annahme der Nutzereingaben (Komplette GUI-Verwaltung)
    - Die ersten Schritte der Nutzereingaben sind in **View**
  - ◊ **Controller:**
    - Wer spielt, ob gerade ein Spiel läuft
    - Schicht zwischen **Model** und **Controller** oft dünn
- ▷ Datenflüsse (Schach):
  - ◊ Von der **View** zum **Model**: Welchen Zug der Spieler gemacht hat
    - Einzig notwendige Information für **Model**: von wo nach wo Figur gezogen
  - ◊ Vom **Model** zur **View**: Ob Spielerzug korrekt war und Ergebnis des **Model**
  - ◊ Vom **Model** zum **Controller**: Ob das Spiel zu Ende ist + Ergebnis
  - ◊ Von **View** zum **Controller**:
    - Drücken des Buttons für neues Spiel
    - Drücken des Buttons für Programmende
  - ◊ Vom **Controller** zum **Model** und **View**:
    - Beginn eines neuen Spiels
    - Anzeige des aktuellen Rankings
  - ◊ Ansonsten sind die Teile völlig unabhängig voneinander
- ▷ Umsetzung in Java (Schach):
  - ◊ Fenster enthält Schachbrett + weitere Buttons
    - von **View** gezeichnet
  - ◊ An jeder GUI-Komponente hängen spezifische Listener
  - ◊ Datenfluss von den Listnern:
    - am Spielbrett (z.B. Canvas): Züge des Spielers → an das **Model**
    - an Buttons für z.B. Spielende → an den **Controller**
    - an Buttons für Änderung der Darstellung → an die **View**
- ▷ Vorteile von MVC:
  - ◊ Änderung einer Komponente → Minimale Änderung der anderen
  - ◊ **View** und **Controller** sind problemlos austauschbar
    - z.B. neue Plattform → Anderes **View**
  - ◊ Mehrere Views gleichzeitig sind möglich
    - Verschieden gestaltet, verschiedene Geräte,..
    - Konsistent aufgrund des **Model** unabhängig von **View**

## Konformität

- ▷ Konformität von Subtypen zu ihren Basistypen
- ▷ Liskov Substitution Principle (LSP):
  - ◇ Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten.
  - ◇ Konstantes Ergebnis, falls statischer und dynamischer Typ ungleich sind
  - ◇ Relevant beim Überschreiben von Methoden
- ▷ Erlaubt nach LSP beim Überschreiben im Subtyp sind:
  - ◇ Anpassung der Funktionalität an die Besonderheiten des Subtyps
    - z.B. `read` von `Reader` / `BufferedReader`
  - ◇ Zusätzliche Funktionalität, die keinen Effekt auf erwartetes Verhalten hat
    - z.B. `Window` und `Frame` // Hinzufügen eines Rahmens
- ▷ Verboten nach LSP beim Überschreiben im Subtyp ist
  - ◇ Änderung Funktionalität, die zu Effekten bei Verwendung des Basistyps führt
    - z.B. ein Subtyp von `List` zählt ab 1 statt ab 0
- ▷ Unterstützung für LSP in Java:
  - ◇ Der Kopf der überschriebenen Methode darf nur im engen Maße abweichen
    - Kette der Zugriffsrechte
    - Verwendung eines Subtyps des Rückgabewerts als Rückgabewert
    - Werfen von Subtypen der original geworfenen Exception
- ▷ Sicherheitslücke um Subtypen von Arrays zuzulassen:
  - ◇ Y ist Subtyp von X

```
1  X[] a = new Y[200];
2  X x = a[150];
3  a[150] = new X();
```
  - ◇ Zeile 1: Zuweisen von ArrayObjekten des Subtyps ist kein Problem
  - ◇ Zeile 2: Lesender Zugriff auf die Komponenten des Arrays auch nicht
  - ◇ Zeile 3: Diese Anweisung geht zwar durch den Compiler, aber ist falsch
  - ◇ Werfen einer `ArrayStoreException` `extends RuntimeException`
- ▷ Methoden, die für abgeleitete Klasse potentiell unpassend sind:
  - ◇ Stichwort: Kreis-Ellipse-Dilemma
  - ◇ In Basisklasse Definition mit `Exception`
    - `...throws UnsupportedOperationException {...}`
    - Abgeleitet von `RuntimeException`, muss also nicht gefangen werden
    - Allerdings gefährlich, da potentiell Programmabbruch
  - ◇ Methode in Subklasse macht dann nichts außer diese `Exception` zu werfen
  - ◇ Interfaces stellen potentielle Lösung dar (`Mixin`)

## 31 Anhang: Polymorphie

### Generelle Einteilung

Bisherige Konzepte	<ul style="list-style-type: none"><li>▷ <b>Racket:</b><ul style="list-style-type: none"><li>◊ Funktionen, die auf versch. Typen mit versch. Operationen arbeiten können<ul style="list-style-type: none"><li>- z.B. <code>foldr</code>, <code>map</code>, ...</li></ul></li></ul></li><li>▷ <b>Java:</b><ul style="list-style-type: none"><li>◊ Ableitungen von Klassen / Implementierung von Interfaces<ul style="list-style-type: none"><li>- Speicherung eines Subtyp-Objekts in Referenz des Basistyps</li><li>- Typ in gewissen Grenzen frei wählbar</li></ul></li><li>◊ Ad-hoc Polymorphie (eher primitiv)<ul style="list-style-type: none"><li>- Methodenüberladung und implizite Konversion</li></ul></li><li>◊ <b>Generics</b><ul style="list-style-type: none"><li>- Typparameter sind die polymorphen Typen</li></ul></li></ul></li></ul>
Einteilung von Konzepten	<ul style="list-style-type: none"><li>▷ Betrachtung von zwei Zeitpunkten<ul style="list-style-type: none"><li>◊ Zeitpunkt der Übersetzung</li><li>◊ Zeitpunkt an dem Ablauf an der Stelle im Quelltext ist</li></ul></li><li>▷ Konformitätsprüfung:<ul style="list-style-type: none"><li>◊ zur Kompilierzeit → statische Prüfung</li><li>◊ zur Laufzeit → dynamische Prüfung</li></ul></li><li>▷ Ansteuerung der korrekten Implementation: (Auswahl der Subroutinen für Typ)<ul style="list-style-type: none"><li>◊ zur Kompilierzeit → statische Bindung</li><li>◊ zur Laufzeit → dynamische Bindung</li></ul></li><li>▷ Funktionen in <b>Racket</b>:<ul style="list-style-type: none"><li>◊ Dynamische Prüfung (wird geprüft, wenn die Operation ausgeführt wird)</li><li>◊ Dynamische Bindung (Steuerung nach erfolgreicher Prüfung)</li></ul></li><li>▷ <b>Java</b> - Klassen ableiten / Interfaces implementieren:<ul style="list-style-type: none"><li>◊ Statische Prüfung (Inspektion des Quelltexts ausreichend)</li><li>◊ Dynamische Bindung (Auswahl welcher Implementation erst beim Aufruf)</li></ul></li><li>▷ <b>Java</b> - Ad-hoc Polymorphie:<ul style="list-style-type: none"><li>◊ Statische Prüfung</li><li>◊ Statische Bindung (Impliziten Konversionen)</li><li>◊ Dynamische Bindung (Methodenüberladung)<ul style="list-style-type: none"><li>- Hier wäre generell aber auch statische Bindung möglich (Java-spezifisch)</li></ul></li></ul></li><li>▷ <b>Java-Generics</b>:<ul style="list-style-type: none"><li>◊ Statische Prüfung (Überprüfung ob Typparameter passend)</li><li>◊ Dynamische Bindung<ul style="list-style-type: none"><li>- Auch hier statische Bindung eigentlich möglich</li></ul></li></ul></li><li>▷ Typische Begriffe:<ul style="list-style-type: none"><li>◊ <b>Duck-Typing</b>: dynamische Prüfung und Bindung</li><li>◊ <b>Subtyppolymorphie</b>: statische Prüfung und dynamische Bindung</li><li>◊ <b>Generizität</b>: statische Prüfung und Bindung</li></ul></li></ul>

### Abstraktion und Polymorphie

Abstraktion	<ul style="list-style-type: none"><li>▷ Chaos an Entitäten gedanklich in geeigneter Form strukturieren<ul style="list-style-type: none"><li>◊ Zuordnung der Entitäten zu passenden Kategorien<ul style="list-style-type: none"><li>- <b>Differenzierung</b></li></ul></li><li>◊ Bei jeder Kategorie:<ul style="list-style-type: none"><li>- Herausfaktorisieren des Gemeinsamen aller Elemente in der Kategorie</li><li>- können auch mehrere gemeinsame, zu trennende Aspekte sein</li><li>- Unterschiedliches für sich stehen lassen</li><li>- <b>Generalisierung</b></li></ul></li></ul></li><li>◊ Beziehungen zwischen den Kategorien herstellen</li></ul>
-------------	--

Beispiel Racket	<ul style="list-style-type: none"> <li>▷ Entitäten: Datenverarbeitungsaufgaben mittels einer Durchlauf durch Liste <ul style="list-style-type: none"> <li>◊ Selbe Standardaufgaben immer wieder</li> </ul> </li> <li>▷ Kategorie: <b>fold</b>, <b>filter</b> und <b>map</b></li> <li>▷ Bei jeder Kategorie: <ul style="list-style-type: none"> <li>◊ Gemeinsamkeit in jeweilige Funktion herausfaktoriert</li> <li>◊ Unterschiede: Unterschiedliche Parameter neben der Liste</li> </ul> </li> <li>▷ Beziehungen: <b>filter</b> und <b>map</b> sind <b>intermediate</b> <ul style="list-style-type: none"> <li>◊ <b>fold</b> ist hingegen <b>terminal</b></li> </ul> </li> </ul>
Beispiel Java	<ul style="list-style-type: none"> <li>▷ <b>Component</b>: <b>Button</b>, <b>Canvas</b>,...</li> <li>◊ Gemeinsame Abstraktion: Arten von Komponenten in GUI</li> <li>◊ Gemeinsamkeiten in <b>Component</b> herausfaktoriert</li> <li>▷ <b>Listener</b>: <b>KeyListener</b>, <b>MouseListener</b>,...</li> <li>◊ Funktionalität der <b>Listener</b> herausfaktoriert</li> <li>◊ Getrennt von der jeweiligen Komponente</li> <li>▷ <b>Event</b>: <b>ActionEvent</b>, <b>KeyEvent</b>,...</li> <li>◊ Auch <b>Event</b> aus dem Konzept <b>Listener</b> ausgelagert</li> </ul>
Abstraktion auf Typebene	<ul style="list-style-type: none"> <li>▷ In logischer Einheit sind ein/mehrere Typen nicht festgelegt</li> <li>▷ Können bei der Nutzung aus Menge von Typen gewählt werden</li> <li>▷ Gewählte Typen müssen verlangte Funktionalitäten bieten</li> <li>▷ Mehrene offene Typen müssen auch gemeinsam korrekt sein</li> </ul>
Polymorphie	<ul style="list-style-type: none"> <li>▷ Oberbegriff für alle Programmierkonzepte, mit denen Abstraktion auf Typebene realisiert werden kann</li> <li>▷ Gründe für Polymorphie: <ul style="list-style-type: none"> <li>◊ Separation of Concerns <ul style="list-style-type: none"> <li>- SoC hat aber natürlich auch viele weitere Gesichtspunkte</li> </ul> </li> <li>◊ Gleichbehandlung von Typen, wo die Unterschiede vernachlässigbar sind</li> <li>◊ Einheit (deren Logik auf versch. Typen passt) nur einmal implementieren <ul style="list-style-type: none"> <li>- Unterschiede dann in ausgelagerten Details</li> </ul> </li> </ul> </li> </ul>
Konzepte von Konformität	<ul style="list-style-type: none"> <li>▷ Konformität ist sehr vielfältig, viele Arten sie abzu prüfen</li> <li>▷ Zusammenfassung bisheriger Stoff dazu: <ul style="list-style-type: none"> <li>◊ <b>Racket</b>: ob Operationen für Operanden definiert sind</li> <li>◊ Suptypppolymorphie: nur Funktionalität des statischen Typs erlaubt</li> <li>◊ Ad-hoc Polymorphie: <ul style="list-style-type: none"> <li>- Methodenüberladung: Überprüfung, ob Signatur einzigartig</li> <li>- Implizite Konversion: Abprüfen eingebauter Regeln</li> </ul> </li> <li>◊ Generizität: unterschiedliche Modelle, sprachabhängig</li> </ul> </li> </ul>

Duck-Typing  
objektorientiert

- ▷ rein dynamische Polymorphie
- ▷ **Reflection** in Java (`java.lang.reflect.*`)
  - ◊ **Java Beans**: Duck-Typing Konzept, das auf **Reflection** beruht
- ▷ Möglichkeiten zur Analyse und den Methodenaufruf eines unbekannten Objekts

```
1 Integer i = 123;
2 String str = "Hello";
3 Class<?> c = Class.forName(nameOfClass);
4 Method m = c.getDeclaredMethod(myMethod",
5           i.getClass(), str.getClass());
6 m.invoke(i, str);
```

  - ◊ Zeile 3: Liefert das **Class**-Objekt für übergebene Klasse zurück
    - Kompletter Name + Package-Pfad als Parameter
    - `java.lang.Class` bietet Funktionalitäten rund um Klassen
    - Zu jeder Klasse **X** existiert ein Objekt von **Class** (`Class<X>`)
    - Existieren auch für primitive Datentypen und für `void`
  - ◊ Zeile 4: Liefert die abgefragte Methode des **Class**-Objekts zurück
    - Erste Parameter ist der Name der Methode
    - beliebig viele weitere Parameter, methodenabhängig
    - Muss genauso viele Parameter enthalten, wie die abgefragte Methode
    - Jeder Parameter ist das **Class**-Objekt des Parametertyps
  - ◊ Zeile 6: Aufruf der abgespeicherten Methode
    - Parameter müssen natürlich übereinstimmen
  - ◊ Duck-Typing Anpassung:
    - Übergabe eines **Object** `obj` an Methode oben
    - `Class<?> c = obj.getClass();` → **dynamische Prüfung**