

FOP Reference Sheet

Jonas Milkovits

Last Edited: 30. 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	12
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

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

1 Collections

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

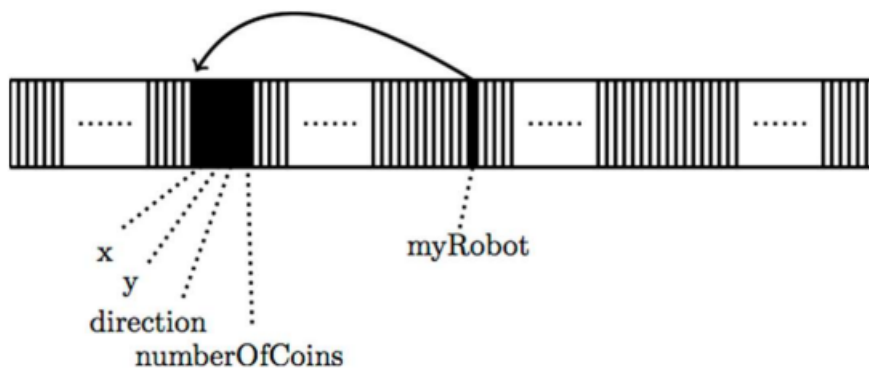
Interface Iterator	<ul style="list-style-type: none"> ▷ Collection und List erben von Interface Iterable ▷ Jede Klasse, die Collection implementiert hat eine eigene Iterator-Klasse ▷ Diese eigene Iterator-Klasse implementiert das Interface Iterator ▷ <code>Collection<Number> c1 = new ArrayList<Number>();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

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

5 Exceptions (java.lang.Exception;)

Exception-Klassen	<ul style="list-style-type: none"> ▷ Alle Klassen, die direkt oder indirekt von java.lang.Exception abgeleitet sind
Exception werfen	<ul style="list-style-type: none"> ▷ <code>throws Exception {...}</code> nach Parameterliste im Methodenkopf ▷ Dies signalisiert, dass die Methode mindestens einen Fehler wirft ▷ Die geworfene Exception muss vom <code>throws</code>-Typ oder Subtyp sein ▷ Auch mehrere Exceptions möglich, mit einem Komma getrennt ▷ Werfen der Exception: <ul style="list-style-type: none"> ◊ z.B.: <code>throw new Exception (No lower case letter!);</code> ◊ Hier wird als Parameter für die Objekterstellung ein String übergeben ▷ <code>throws</code>: <ul style="list-style-type: none"> ◊ Führt zur Beendigung der Methode ◊ Liefert das geworfene Exception-Objekt zurück
Exception fangen	<ul style="list-style-type: none"> ▷ Bei Methoden, die Exceptions werfen, wird ein <code>try-catch</code>-Block benötigt ▷ Aufbau: <ul style="list-style-type: none"> ◊ Methoden, die Exceptions werfen in <code>try {...}</code> aufrufen ◊ Falls Exception auftritt wird <code>catch (Exception exc) {...}</code> aufgerufen ◊ <code>catch</code> muss direkt im Anschluss nach <code>try</code> stehen ◊ Falls kein Fehler auftritt, wird <code>catch</code> übersprungen ◊ Das Programm wird dann normal weiter ausgeführt ▷ Es sind auch mehrere <code>catch</code>-Blöcke mit versch. Parametern möglich ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getMessage(); // Returns the error message as a string</code> ◊ <code>printStackTrace(); // Ausgabe des Call-Stacks</code> ▷ Alle möglichen Exceptions müssen durch den <code>catch</code>-Block abgedeckt sein ▷ Falls Exception zu mehreren <code>catch</code>-Blöcken 'passt', wird der Erste ausgeführt <ul style="list-style-type: none"> ◊ Deswegen Reihung der <code>catch</code>-Blöcke von Subtyp nach Supertyp ▷ Auch mehrere Exceptions in einem <code>catch</code>-Block möglich mit <code> </code>
Weiterreichen	<ul style="list-style-type: none"> ▷ Weiterreichen der Fehlermeldung durch <code>throws</code> im Methodenkopf möglich ▷ Kein <code>try-catch</code>-Block notwendig ▷ Main-Methode kann z.B. keine Exceptions weiterreichen
<code>try-with-resources</code>	<ul style="list-style-type: none"> ▷ Für Ressourcen, die unbedingt wieder geschlossen werden müssen ▷ Öffnung der Ressource in runden Klammern: <code>try (Printer p =...) {...}</code> ▷ Mehrere Ressourcen möglich, getrennt durch Semikolon
Runtime Exceptions	<ul style="list-style-type: none"> ▷ Ausnahme zu <code>try</code>-Blöcken ▷ Exceptions von java.lang.RuntimeException und Subtypen ▷ z.B.: <code>IndexOutOfBoundsException</code>, <code>NullPointerException</code> ▷ Grund: Vermeidung von dauerenden <code>try</code>-Blöcken
Throwable und Error	<ul style="list-style-type: none"> ▷ Exception und Error sind beide von <code>Throwable</code> abgeleitet ▷ Alle drei befinden sich im Paket java.lang ▷ Error: <ul style="list-style-type: none"> ◊ Werden geworfen, falls Fehlerbehandlung keinen Sinn macht ◊ Programmabbruch als Ausweg ▷ <code>AssertionError</code>: <ul style="list-style-type: none"> ◊ <code>throw new AssertionError("Bad!");</code> ◊ Kurzform: <code>assert x == 2: "Bad!";</code> ◊ Wichtig: Bedingung muss negiert werden! ◊ Assertanweisungen sinnvoll, da kurz und übersichtlich ◊ Können zusätzlich vom Compiler an- und abgeschaltet werden ◊ z.B.: Verwendung für Tests für Methoden und späteres Abschalten ▷ Solche Tests werden White-Box-Tests genannt
Exceptions aus Lambda-Ausdrücken	<ul style="list-style-type: none"> ▷ Gute Verwendung, falls die funktionale Methode eines Interface Fehler wirft ▷ <code>MyFuncIntf fct = (m,n) -></code> <pre> {if (n == 0) throw new Exception(); return m/n;}; </pre> ▷ Funktionale Methode könnte führt Berechnung aus ▷ Falls <code>n == 0</code> wird jedoch eine Exception geworfen

6 Fehler

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

7 Files

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

Klasse Files	<ul style="list-style-type: none"> ▷ Aus Package <code>java.nio.file</code> ▷ Nützliche Sammlung von Klassenmethoden rund um Dateien ▷ Methoden: <ul style="list-style-type: none"> ◇ <code>lines</code> // <code>Files.lines(path)</code>; <ul style="list-style-type: none"> - Öffnet Datei an übergebenem Pfad - Liefert einen Stream von Strings, ein String pro Zeile - Zeilenende durch <code>"file.separator"</code> gekennzeichnet - <code>IOException</code>, falls Problem beim Öffnen der Datei (<code>java.io</code>) ◇ <code>exists</code> // <code>Files.exists(path)</code>; <ul style="list-style-type: none"> - <code>true</code>, wenn es dort Datei/Verzeichnis gibt ◇ <code>isReadable(path)</code> <ul style="list-style-type: none"> - Fragt lesende Zugriffsrechte ab ◇ <code>isWritable(path)</code> <ul style="list-style-type: none"> - Fragt schreibende Zugriffsrechte ab ◇ <code>isRegularFile(path)</code> <ul style="list-style-type: none"> - <code>true</code>, falls es eine reguläre Datei ist (kein Verzeichnis) ◇ <code>isDirectory(path)</code> <ul style="list-style-type: none"> - <code>true</code>, falls es ein Verzeichnis ist ◇ <code>size(path)</code> // <code>long size = Files.size(path)</code>; <ul style="list-style-type: none"> - Fragt die Größe der Datei ab - <code>long</code>, da die Dateigröße oft nicht in <code>int</code> passt ◇ <code>createFile(path)</code> <ul style="list-style-type: none"> - Richtet Datei an der übergebenen Stelle ein ◇ <code>copy(path1, path2)</code> <ul style="list-style-type: none"> - Kopieren von Pfad 1 nach Pfad 2 ◇ <code>move(path1, path2)</code> <ul style="list-style-type: none"> - Umbenennen einer Datei, oft auch Bewegen genannt ◇ <code>delete(path)</code> <ul style="list-style-type: none"> - Entfernen einer Datei - <code>NoSuchElementException</code>, falls nicht vorhanden ◇ <code>deleteIfExists(path)</code> <ul style="list-style-type: none"> - Falls das Objekt nicht existiert, passiert garnichts
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<String> 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"> ▷ <code>try-with-resources</code> wird für Interface <code>AutoCloseable</code> verwendet
Bytedaten	<ul style="list-style-type: none"> ▷ Direkt, ohne Bezug zu Streams ▷ Klassen und Interfaces finden sich in <code>java.io</code> ▷ Byteweise Verarbeitung sinnvoll für Audio oder Bilddateien, nicht für Text ▷ Wird aber meist durch Bibliotheken oder Ähnliches gehandhabt
Bytedaten lesen	<ul style="list-style-type: none"> ▷ Verwendung eines <code>InputStream</code>-Objekts ▷ <code>InputStream</code> abstrakt, deswegen nur Subtypen z.B.: <code>FileInputStream</code> <ul style="list-style-type: none"> ◇ <code>FileInputStream</code> nutzt den Namen der Datei als String im Konstruktor ▷ Methoden: <ul style="list-style-type: none"> ◇ <code>read()</code> <ul style="list-style-type: none"> - Liest nächstes Byte in ein <code>int</code> - Überprüfung, ob -1 um zu prüfen, ob Dateiende erreicht ist ▷ Beispiel: <pre> 1 FileInputStream in = new FileInputStream (fileName); 2 int n = in.read(); 3 if (n == 1) return; </pre>

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

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

8 Functional Interfaces und Lambda-Ausdrücke

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

9 Graphical User Interface

Window Manager	<ul style="list-style-type: none"> ▷ Systemprozess, der permanent im Hintergrund als Service läuft ▷ Stellt generelle, anwendungsunspezifische Funktionalitäten zur Verfügung <ul style="list-style-type: none"> ◊ Öffnen, Schließen, Ikonifizieren, Größe ändern ◊ Rahmen um Fenster, Bildschirmhintergrund
Klasse Frame	<ul style="list-style-type: none"> ▷ Abgeleitet von <code>java.awt.Window</code>; (<code>awt</code> = abstract window toolkit) ▷ Im Gegensatz zu <code>Window</code> aber mit Rahmen (vom <code>Window Manager</code> verwaltet) ▷ Beispielkonstruktor: <code>Frame frame = new Frame(string); // Fenstertitel</code> ▷ Vorhergehensweise: <ul style="list-style-type: none"> ◊ Erstellung einer Klasse, die <code>Frame</code> erweitert ◊ Hinzufügen von Funktionalitäten ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>setVisible(boolean b)</code> <ul style="list-style-type: none"> - <code>Frame</code> ist entweder sichtbar oder unsichtbar - Standardmäßig unsichtbar - Erst Fenster aufbauen, dann sichtbar machen ◊ <code>setBackground(Color bgColor)</code> <ul style="list-style-type: none"> - Setzt die Hintergrundfarbe des Fensters ◊ <code>dispose()</code> <ul style="list-style-type: none"> - Alle Ressourcen des Fensters und der Bestandteile werden freigegeben ◊ <code>setExtendedState(int state)</code> <ul style="list-style-type: none"> - Setzt den Status des Fensters - <code>ICONIFIED</code>: Ikonifiziert das Fenster - <code>NORMAL</code>: Deikonifiziert das Fenster - <code>MAXIMIZED_HORIZ</code>: Ausbreitung auf gesamte Horizontale ◊ <code>add(Component comp)</code> <ul style="list-style-type: none"> - Fügt den übergebenen Komponenten zum <code>Frame</code> hinzu
Komponenten	<ul style="list-style-type: none"> ▷ Eigene Klasse für jede Komponente ▷ Alle Klassen oder Interfaces aus <code>java.awt</code>, falls nicht anders gesagt ▷ Werden mithilfe von <code>add(Component comp)</code> zum Fenster hinzugefügt
Button	<ul style="list-style-type: none"> ▷ Konstruktor: <code>Button(String label) // Text auf dem Button</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>setFont(Font f)</code> <ul style="list-style-type: none"> - Zum Setzen der Schriftart - Konstruktor <code>Font</code>: <code>Font(String name, int style, int size)</code> ◊ <code>addActionListener(ActionListener l)</code> <ul style="list-style-type: none"> - Fügt den übergebenen <code>ActionListener</code> hinzu - Bei jedem Klick wird <code>actionPerformed</code> des <code>Listeners</code> aufgerufen - Auch mehrere möglich - Automatische Einrichtung des <code>Event Dispatch Thread</code> ◊ <code>setLabel(String label)</code> <ul style="list-style-type: none"> - Setzt den Titel des <code>Button</code>

Interface ActionListener	<ul style="list-style-type: none"> ▷ Zugehörig zu <code>Button</code> ▷ Aus Package <code>java.awt.event</code> ▷ Funktionales Interface ▷ Funktionale Methode <code>actionPerformed (ActionEvent event)</code> ▷ Vorgehensweise: <ul style="list-style-type: none"> ◊ Erstellen einer eigenen Klasse, die <code>ActionListener</code> implementiert ◊ Erstellen relevanter Attribute und Konstruktor für gegebenen Fall ◊ Implementieren der Methode <code>actionPerformed (ActionEvent event)</code> ◊ Erstellen eines Objekts unserer Klasse <ul style="list-style-type: none"> - <code>ActionListener listener = new MyListener(frame);</code> ◊ Hinzufügen des Listener zum <code>Button</code> <ul style="list-style-type: none"> - <code>button.addActionListener(listener);</code> ▷ Alternativ: <ul style="list-style-type: none"> ◊ Erstellung des Listener in der Subklasse des Frame <ul style="list-style-type: none"> - Keine Frame-Übergabe notwendig - z.B.: als <code>private</code>-Klasse (Stichwort: <code>nested classes</code>) ▷ Da <code>Functional Interface</code>: Lambda-Ausdruck <ul style="list-style-type: none"> ◊ <code>button.addActionListener((e) -> {System.out.print("Hello");})</code>
Klasse <code>ActionEvent</code>	<ul style="list-style-type: none"> ▷ Übergebener Parameter bei <code>actionPerformed</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getWhen()</code> <ul style="list-style-type: none"> - Gibt die Uhrzeit des Geschehnisses als <code>long</code> zurück - Nützlich: <code>java.sql.Timestamp</code> - <code>Timestamp stamp = new Timestamp (event.getWhen());</code> - Methoden: <code>stamp.getHour(); stamp.getMinute();</code>
Übersicht Listener und Events	<ul style="list-style-type: none"> ▷ Listener-Interface ↔ Event-Klasse ▷ <code>KeyListener</code> ↔ <code>KeyEvent</code> ▷ <code>MouseListener</code> ↔ <code>MouseEvent</code> ▷ <code>MouseMotionListener</code> ↔ <code>MouseEvent</code> ▷ <code>MouseWheelListener</code> ↔ <code>MouseWheelEvent</code> ▷ <code>WindowFocusListener</code> ↔ <code>WindowEvent</code> ▷ <code>WindowListener</code> ↔ <code>WindowEvent</code> ▷ <code>WindowStateListener</code> ↔ <code>WindowEvent</code> ▷ Hinzufügen: <ul style="list-style-type: none"> ◊ <code>addKeyListener(...)</code> ◊ <code>addMouseListener(...)</code> ◊ <code>addWindowListener(...)</code>
Adapter	<ul style="list-style-type: none"> ▷ Verwendung von Adaptern, wenn passendes Interface nicht <code>functional</code> ist <ul style="list-style-type: none"> ◊ z.B. Interface <code>KeyListener</code>, <code>MouseListener</code>,... ◊ Diese Interfaces besitzen mehrere Methoden ▷ Adapter sind Klassen und bestehen zu jedem Listener-Interface <ul style="list-style-type: none"> ◊ z.B.: <code>KeyAdapter</code>, <code>MouseAdapter</code> ◊ Diese Adapter implementieren das dazugehörige Interface ◊ Die Methoden werden jedoch leer gelassen ▷ Vorteil vom Adapter: <ul style="list-style-type: none"> ◊ Nicht alle Methoden müssen implementiert werden ◊ Nur die genutzten Methoden (z.B.: <code>keyPressed()</code>) werden implementiert ▷ Verwendung: <ul style="list-style-type: none"> ◊ Erweitern der eigenen Listener-Klasse mit Adapter ◊ z.B.: <code>public class MyKeyListener extends KeyAdapter {...}</code>

Interface KeyListener	<ul style="list-style-type: none"> ▷ Abhören der Tastatur ▷ Erstellen eigener Klasse, die die Klasse <code>KeyAdapter</code> (siehe Adapter) erweitert ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void keyPressed (KeyEvent event)</code> - Wird beim Herunterdrücken einer Taste ausgeführt ◊ <code>public void keyReleased (KeyEvent event)</code> - Wird beim Loslassen einer Taste ausgeführt ◊ <code>public void keyTyped (KeyEvent event)</code> - Wird beim Antippen einer Taste ausgeführt
Klasse KeyEvent	<ul style="list-style-type: none"> ▷ Übergebener Parameter bei z.B.: <code>keyPressed</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getKeyCode()</code> - Liefert die Kodierung der gedrückten Taste zurück ▷ Klassenkonstanten für jede Taste: <ul style="list-style-type: none"> ◊ z.B.: <code>KeyEvent.VK_A</code> // Buchstabe A ◊ z.B.: <code>KeyEvent.VK_COLON</code> // Doppelpunkt ◊ z.B.: <code>KeyEvent.VK_BACKSPACE</code> // Backspace Taste ▷ 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>
Interface MouseListener	<ul style="list-style-type: none"> ▷ Abhören der Maus ▷ Erstellen eigener Klasse, die die Klasse <code>MouseAdapter</code> erweitert <ul style="list-style-type: none"> ◊ <code>MouseAdapter</code> implementiert alle drei Mouse-Interfaces ◊ <code>MouseListener</code>, <code>MouseMotionListener</code>, <code>MouseWheelListener</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void mouseClicked (MouseEvent event)</code> - Wird beim kurzen Klicken der Maustaste ausgeführt ◊ <code>public void mousePressed (MouseEvent event)</code> - Wird beim Herunterdrücken der Maustaste ausgeführt ◊ <code>public void mouseReleased (MouseEvent event)</code> - Wird beim Loslassen der Maustaste ausgeführt ◊ <code>public void mouseEntered (MouseEvent event)</code> - Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich betritt ◊ <code>public void mouseExited (MouseEvent event)</code> - Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich verlässt
Interface MouseMotionListener	<ul style="list-style-type: none"> ▷ Abhören der Mausbewegung ▷ Methoden sind auch in Klasse <code>MouseAdapter</code> enthalten ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void mouseDragged (MouseEvent event)</code> ◊ <code>public void mouseMoved (MouseEvent event)</code>
Interface MouseWheelListener	<ul style="list-style-type: none"> ▷ Abhören der Mousradbewegung ▷ Methoden sind auch in Klasse <code>MouseAdapter</code> enthalten ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void mouseWheelMoved (MouseWheelEvent event)</code>

Klasse MouseEvent	<ul style="list-style-type: none"> ▷ Übergebener Parameter bei z.B.: <code>mouseClicked</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getButton()</code> <ul style="list-style-type: none"> - Liefert die gedrückte Taste zurück ◊ <code>getX()</code> <ul style="list-style-type: none"> - Liefert x-Koordinate abhängig vom Ursprung des Bereichs ◊ <code>getY()</code> <ul style="list-style-type: none"> - Liefert y-Koordinate abhängig vom Ursprung des Bereichs ▷ Klassenkonstanten für Maustasten: <ul style="list-style-type: none"> ◊ <code>MouseEvent.BUTTON1</code> ◊ <code>MouseEvent.BUTTON2</code>
Klasse MouseEvent	<ul style="list-style-type: none"> ▷ Übergebener Parameter bei z.B.: <code>mouseWheelMoved</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getWheelRotation()</code> <ul style="list-style-type: none"> - Liefert die Anzahl der gedrehten Ticks"
Interface WindowListener	<ul style="list-style-type: none"> ▷ Abhören von Fensteraktionen ▷ Erstellen eigener Klasse, die die Klasse <code>WindowAdapter</code> erweitert <ul style="list-style-type: none"> ◊ <code>WindowAdapter</code> implementiert alle drei Window-Interfaces ◊ <code>WindowListener</code>, <code>WindowStateListener</code>, <code>WindowFocusListener</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void windowOpened (WindowEvent event)</code> ◊ <code>public void windowClosing (WindowEvent event)</code> ◊ <code>public void windowClosed (WindowEvent event)</code> ◊ <code>public void windowClosed (WindowEvent event)</code> ◊ <code>public void windowDeactivated (WindowEvent event)</code> ◊ <code>public void windowIconified (WindowEvent event)</code> ◊ <code>public void windowDeiconified (WindowEvent event)</code>
Interface WindowStateListener	<ul style="list-style-type: none"> ▷ Abhören des Status des Fensters ▷ Methoden sind auch in <code>WindowAdapter</code> vorhanden ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void windowStateChanged (WindowEvent event)</code>
Interface WindowFocusListener	<ul style="list-style-type: none"> ▷ Abhören des Fokus im Bezug auf das Fenster ▷ Methoden sind auch in <code>WindowAdapter</code> vorhanden ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>public void windowGainedFocus (WindowEvent event)</code> ◊ <code>public void windowLostFocus (WindowEvent event)</code>

Klasse Canvas	<ul style="list-style-type: none"> ▷ abgegrenzte Zeichenfläche in einem Fenster ▷ Vorhergehensweise: <ul style="list-style-type: none"> ◊ Erstellung eigener Subtyp-Klasse von <code>Canvas</code> ◊ Implementieren der Methode <code>public void paint (Graphics graphics)</code> ◊ Füllen der Methode mit eigener Zeichenlogik ◊ Verwendung von <code>java.awt.Graphics</code>; ◊ Hinzufügen zum <code>Frame</code> mithilfe von <code>add</code> ▷ Beleuchtung nützlicher Aspekte von <code>Graphics</code>: ▷ <code>FontMetrics</code> <ul style="list-style-type: none"> ◊ Informationen über festgelegte Schriftart und Schriftgröße ◊ Abfrage: <ul style="list-style-type: none"> - <code>FontMetrics fontM = graphics.getFontMetrics();</code> ◊ Abfrage der maximalen Stringhöhe: <ul style="list-style-type: none"> - <code>int maxHeight = fontM.getMaxAscent() + fontM.getMaxDescent();</code> - Methoden geben maximalen Abstand von der Basislinie des Textes an ◊ Abfrage der Stringbreite von gegebenem String: <ul style="list-style-type: none"> - <code>int widthStr = fontMetrics.stringWidth(string);</code> ▷ Abfrage des Zeichenfensters als Rechteck: <ul style="list-style-type: none"> - <code>Rectangle area = graphics.getClipBounds();</code> - <code>x</code> und <code>y</code> geben den Ursprung an - <code>width</code> und <code>height</code> die Breite und Höhe ▷ Einige Methoden von <code>Graphics</code> <ul style="list-style-type: none"> ◊ <code>setColor(Color color)</code> ◊ <code>fillOval(...)</code> ◊ <code>drawOval(...)</code> ◊ <code>drawString(...)</code>
Klasse Checkbox	<ul style="list-style-type: none"> ▷ Kleiner Button (Pin) mit etwas Text ▷ Zwei Zustände: An oder Aus ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>Checkbox(String label) // Titel der Checkbox</code> ◊ <code>Checkbox</code> standardmäßig aus ▷ Benötigt ein Objekt vom Typ <code>ItemListener</code> (siehe unten) <ul style="list-style-type: none"> ◊ <code>ItemListener item = new MyItemListener(checkbox,...);</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>isSelected()</code> <ul style="list-style-type: none"> - <code>true</code>, wenn die <code>Checkbox</code> an ist ◊ <code>setLabel(string);</code> <ul style="list-style-type: none"> - Setzt den Titel der <code>Checkbox</code>
Interface ItemListener	<ul style="list-style-type: none"> ▷ Verwendung bei <code>Checkbox</code> und <code>Choice</code> ▷ Funktionales Interface ▷ Funktionale Methode <code>itemStateChanged (ItemEvent event)</code> ▷ Vorhergehensweise analog zu <code>ActionListener</code> <ul style="list-style-type: none"> ◊ Erstellung neuer Klasse, die <code>ItemListener</code> implementiert

Klasse Choice	<ul style="list-style-type: none"> ▷ Repräsentiert ein Auswahlmenü ▷ Verwendet auch das Interface <code>ItemListener</code> ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>Choice choice = new Choice();</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>add(String)</code> <ul style="list-style-type: none"> - Hinzufügen neuer Auswahlen - Startet bei Index 0 ◊ <code>select(int)</code> <ul style="list-style-type: none"> - Legt eine Auswahl als Standard fest - Übergabe des Index als <code>int</code> ◊ <code>getSelectedItem()</code> <ul style="list-style-type: none"> - Liefert den ausgewählten String zurück ◊ <code>getSelectedIndex()</code> <ul style="list-style-type: none"> - Liefert Index der aktiven Auswahl
Klasse Label	<ul style="list-style-type: none"> ▷ Nicht durch User interagierbares Rechteck mit Text ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>Label(String text) // Labeltext</code> ▷ Wartet auf Events bei anderen Entitäten ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>setAlignment(int alignment)</code> <ul style="list-style-type: none"> - Auswahl der Zentrierung des Textes - Parameter: <code>Label.CENTER</code>, <code>Label.RIGHT</code>, <code>Label.LEFT</code> ◊ <code>setBackground(Color c)</code> <ul style="list-style-type: none"> - Setzen der Hintergrundfarbe ◊ <code>setText(String text)</code> <ul style="list-style-type: none"> - Setzt den Text des Label - z.B.: Aufruf beim Drücken eines Button
Klasse List	<ul style="list-style-type: none"> ▷ Auswahlmenü ▷ Aus <code>java.awt</code>, nicht <code>java.util</code> ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>List(int rows, boolean multipleMode)</code> ◊ <code>rows</code> gibt die maximale Anzahl der zugleich angezeigten Menüpunkte an ◊ Anzahl der Möglichkeiten größer als <code>rows</code> → Scrollbar ◊ <code>multipleMode</code>: Auswahl mehrerer Menüpunkte ermöglichen ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>add(String item)</code> <ul style="list-style-type: none"> - Hinzufügen neuer Menüpunkte ◊ <code>getSelectedIndexes()</code> <ul style="list-style-type: none"> - Liefert die Indizes der ausgewählten Punkte ◊ <code>setMultipleMode(boolean b)</code> <ul style="list-style-type: none"> - De-/Aktivieren der Mehrfachauswahl
Klasse Scrollbar	<ul style="list-style-type: none"> ▷ Werden meist automatisch hinzugefügt (<code>List</code>) ▷ z.B.: Erstellung eines eigenen Schiebereglers ▷ Benötigt ein Objekt vom Typ <code>AdjustmentListener</code> (siehe unten) <ul style="list-style-type: none"> ◊ z.B.: <code>AdjustmentListener adjust = new MyAdjustListener(frame);</code> ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>Scrollbar(int orientation, int value, int visible, int minimum, int maximum)</code> ◊ <code>orientation</code>: <code>Scrollbar.VERTICAL</code>, <code>Scrollbar.HORIZONTAL</code> ◊ <code>value</code>: Startwert der Scrollbar ◊ <code>visible</code>: Größe des scrollbaren Balkens ◊ <code>minimum</code>: Minimal einstellbarer Wert ◊ <code>maximum</code>: Maximal einstellbarer Wert

Interface AdjustmentListener	<ul style="list-style-type: none"> ▷ Verwendung bei Scrollbar ▷ Funktionales Interface ▷ Funktionale Methode: <code>adjustmentValueChanged (AdjustmentEvent event)</code> ▷ Vorhergehensweise analog zu ActionListener <ul style="list-style-type: none"> ◊ Erstellung neuer Klasse, die AdjustmentListener implementiert
Klasse Adjustmentevent	<ul style="list-style-type: none"> ▷ Übergebener Parameter bei <code>adjustmentValueChanged</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getValue()</code> <ul style="list-style-type: none"> - Liefert den neuen Wert der Scrollbar
Klasse Textfield	<ul style="list-style-type: none"> ▷ Zeile, vom Nutzer schreibbar ▷ z.B.: Benutzername, Passwort, etc.. ▷ Benötigt ein Objekt vom Typ KeyListener (siehe oben) ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>TextField(int columns)</code> ◊ <code>columns</code> gibt die Zeichenzahl in der Zeile an ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>setEchoChar(char c)</code> <ul style="list-style-type: none"> - Anzeige der eingegebenen Zeichen mit anderem Zeichen z.B.: '*' - Rückgängig machen: <code>field.setEchochar((char) 0);</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>getText()</code> <ul style="list-style-type: none"> - Liefert den eingegebenen Text als String
Klasse TextArea	<ul style="list-style-type: none"> ▷ Eingabebereich über mehrere Zeilen ▷ z.B.: Verwendung eines Objekts des Typs FocusListener ▷ Konstruktor: <ul style="list-style-type: none"> ◊ <code>TextArea(String text, int rows, int columns, int scrollbars)</code> ◊ <code>text</code>: Text, falls Bereich leer und nicht im Mausfokus ◊ <code>scrollbars</code>: Legt die Art der Scrollbar fest <ul style="list-style-type: none"> - <code>Scrollbar.BOTH</code>, <code>Scrollbar.HORIZONTAL_ONLY</code> - <code>Scrollbar.NONE</code>, <code>Scrollbar.VERTICAL_ONLY</code> ◊ <code>rows</code>: Anzahl der Zeilen ◊ <code>columns</code>: Breite der Zeilen ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>setText(String t)</code> <ul style="list-style-type: none"> - Setzt den Text des Textfeldes ◊ <code>getText()</code> <ul style="list-style-type: none"> - Liefert den geschriebenen Text als String ▷ Leerer Text: <code>("")</code> <ul style="list-style-type: none"> ◊ z.B.: Vergleich mit momentanem Text mit <code>equals</code>
Interface FocusListener	<ul style="list-style-type: none"> ▷ Verwendung bei TextArea ▷ Kein funktionales Interface, trotzdem keine Adapter-Klasse ▷ Vorhergehensweise analog zu ActionListener ▷ Im Gegensatz zu WindowFocusListener auch für einzelne Komponenten ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>focusGained(FocusEvent e)</code> <ul style="list-style-type: none"> - Wird ausgeführt, wenn eine Komponente den Tastaturfokus erhält ◊ <code>focusLost(FocusEvent e)</code> <ul style="list-style-type: none"> - Wird ausgeführt, wenn eine Komponente den Tastaturfokus verliert

Hierarchie graphischer Komponenten	<ul style="list-style-type: none"> ▷ Vom <code>java.awt.Component</code> direkt abgeleitet: <ul style="list-style-type: none"> ◊ <code>Button</code> ◊ <code>Canvas</code> ◊ <code>Checkbox</code> ◊ <code>Choice</code> ◊ <code>Label</code> ◊ <code>List</code> ◊ <code>Scrollbar</code> ◊ <code>TextComponent</code> // Supertyp von <code>TextArea</code> und <code>TextField</code> ◊ <code>Container</code> ▷ Von <code>Container</code> direkt abgeleitet: <ul style="list-style-type: none"> ◊ <code>Window</code> ▷ Von <code>Window</code> direkt abgeleitet: <ul style="list-style-type: none"> ◊ <code>Frame</code>
Klasse <code>Component</code>	<ul style="list-style-type: none"> ▷ Die meisten Methoden sind hier definiert, aber nicht implementiert <ul style="list-style-type: none"> ◊ z.B.: <code>setVisible(boolean b)</code>, <code>setFont(Font f)</code>,... ◊ Die Methoden werden in den Komponentenklassen dann implementiert
Klasse <code>Container</code>	<ul style="list-style-type: none"> ▷ Fasst mehrere Komponenten zu einer zusammen ▷ Hinzufügen von <code>Buttons</code>,..., <code>Windows</code>, <code>Frames</code>, <code>Containern</code> möglich ▷ Wichtig: Hinzufügen von <code>Container</code> möglich <ul style="list-style-type: none"> ◊ Ähnliche Struktur wie ein Ordnerverzeichnis ◊ z.B.: <code>Frame</code> in einem <code>Frame</code> ▷ Methoden: <ul style="list-style-type: none"> ◊ <code>paint (Graphics graphics)</code> <ul style="list-style-type: none"> - In <code>Component</code> definiert, hier überschrieben - Ruft <code>paint</code> für jeden enthaltenen Komponenten auf ◊ <code>add (Component comp)</code> <ul style="list-style-type: none"> - Hinzufügen einer Komponente zum <code>Container</code> ◊ <code>add (Component comp, Object constraints)</code> <ul style="list-style-type: none"> - Steuerung der Position mithilfe des zweiten Parameters - Weiteres bei <code>LayoutManager</code> ◊ <code>setLayout (LayoutManager manager)</code> <ul style="list-style-type: none"> - Setzen des <code>LayoutManager</code> - Dieser steuert die Platzierung der Komponenten - Jeder <code>Container</code> hat zu jedem Zeitpunkt einen <code>LayoutManager</code> ◊ <code>validate()</code> <ul style="list-style-type: none"> - Aktualisierung nach z.B.: Größenänderung

Klasse
LayoutManager

- ▷ Wird bei Erstellung eines Containers oder Subtypes automatisch eingerichtet
 - ◊ Standardklasse für für Window und Frame ist BorderLayout
- ▷ BorderLayout
- ▷ Einteilung des Fensters in fünf Bereiche
 - ◊ NORTH, EAST, SOUTH, WEST, CENTER
 - ◊ Mögliche Positionen als Klassenkonstanten vordefiniert
 - z.B.: BorderLayout.NORTH, BorderLayout.CENTER,...
 - ◊ Verwendung bei add (Component comp, Object constraints)
 - z.B.: frame.add (comp1, BorderLayout.NORTH);
 - Ohne Wahl der Position (normales add): CENTER als Standard
- ▷ BorderLayout an sich für das Fenster an sich meist die richtige Wahl
 - ◊ Aber nicht unbedingt für Container innerhalb eines Fensters
- ▷ BoxLayout
 - ◊ Anlegen in einer Reihe nacheinander
 - ◊ Wahl ob vertikal oder horizontal im Konstruktor
- ▷ GridLayout
 - ◊ Matrixartiges Anlegen (wie Telefonschaltplan)
 - ◊ Anzahl Zeilen und Spalten im Konstruktor festgelegt
- ▷ BorderLayout, BoxLayout und GridLayout:
 - ◊ Passen Größe der Komponenten anhand der Gesamtsituation an
 - ◊ Nicht unbedingt passendste Größe für Komponente
- ▷ FlowLayout
 - ◊ Anlegen in einer Zeile nebeneinander
 - Anfangen einer Zeile, falls die alte voll ist
 - ◊ Wählt automatisch die bestmögliche Größe für Komponenten
 - Abfrage über getPreferredSize()
- ▷ CardLayout
 - ◊ Zeigt Komponenten nicht alle gleichzeitig, sondern nacheinander
 - ◊ Navigation: first, last, next, previous
- ▷ validate()
 - ◊ Notwendig zur Aktualisierung von sichtbaren Fenstern
 - ◊ Wann:
 - Ändern der Anzahl von Komponenten
 - Ändern der Größe von Komponenten (auch Schrift)

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

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

11 Graphics (java.awt.Graphics;)

Applet	<ul style="list-style-type: none"> ▷ leichtgewichtige Variante an Graphikprogrammen ▷ <code>import java.awt.Applet;</code> ▷ 1. Erstellen eigener Applet-Klasse (<code>extends Applet</code>) ▷ 2. Überschreiben der Methode <code>paint</code> <pre>public void paint (Graphics graphics) {...}</pre> <p>Klasse <code>Graphics</code> verknüpft Programm mit Zeichenfläche</p> ▷ 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> ▷ 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>
GeomShape2D	<ul style="list-style-type: none"> ▷ Abstrakte Klasse (Methode <code>paint</code> ist abstrakt) ▷ Attribute: <pre>int positionX; int positionY; int rotationAngle; int transparencyValue; Color boundaryColor; Color fillColor;</pre> ▷ Subklassen: <code>Rectangle</code>, <code>Circle</code>, <code>StraightLine</code>

12 Interfaces

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

13 JUnit-Tests

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

14 Klassen

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

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

15 Konversionen

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

16 Methoden

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

17 Optional (java.lang.Optional;)

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

18 Packages und Zugriffsrechte

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

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

23 String (java.lang.String)

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

24 Syntax

Keywords	<ul style="list-style-type: none"> ▷ Können nur an bestimmten Stellen im Code stehen ▷ z.B. <code>class</code>, <code>import</code>, <code>public</code>, <code>while</code>,...
Identifizier	<ul style="list-style-type: none"> ▷ Namen für Klassen, Variablen, Methoden,.. ▷ Erstes Zeichen darf keine Ziffer sein ▷ Keine Keywords als Identifizier ▷ Identifizier sind case-sensitive
Konventionen	<ul style="list-style-type: none"> ▷ Variablen / Methoden beginnen mit Kleinbuchstaben (<code>testInt</code>) ▷ Klassen beginnen mit Großbuchstaben (<code>testClass</code>) ▷ Wortanfänge im Inneren mit Großbuchstaben ▷ Konstanten bestehen aus <code>_</code> und Großbuchstaben (<code>CENTS_PER_EURO</code>) ▷ Packagenamen nur aus Kleinbuchstaben und <code>_</code> bei unzulässigen Zeichen ▷ Boolesche Bestandteile: Prädikate (<code>isGreen</code>) ▷ Andere Bestandteile mit Wert: Beschreibung des Wertes ▷ Subroutinen ohne Rückgabe: Imperativ (<code>fill0val</code>)
Kommentare	<ul style="list-style-type: none"> ▷ <code>//</code> Einzelne Zeile ▷ <code>/*...*/</code> Mehrere Zeilen ▷ <code>/**...*/</code> Erzeugung von Javadoc
Javadoc	<ul style="list-style-type: none"> ▷ Erzeugung mithilfe von <code>/**</code> und Enter ▷ Bei Methodenköpfen: <ul style="list-style-type: none"> ◊ <code>@param x the dividend</code> ◊ <code>@return x divided by x</code> ◊ <code>@throws class IndexOutOfBoundsException if c is not an int</code> ▷ Bei Quelldateien: <ul style="list-style-type: none"> ◊ <code>@author</code> ◊ <code>@version</code>
Rechtsausdrücke	<ul style="list-style-type: none"> ▷ Haben Typ und Wert ▷ z.B.: <code>2*3+1</code>
Linksausdrücke	<ul style="list-style-type: none"> ▷ Verweisen auf Speicherstellen ▷ z.B.: <code>int n</code>

25 Threads

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

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

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);<v></code>
Overwrite	▷ Methoden in Subklassen können auch neu geschrieben werden <ul style="list-style-type: none"> ◊ Die Implementation der Superklasse wird sozusagen überschrieben ▷ Selber Name und Parameterliste notwendig ▷ Signatur der Methoden muss identisch sein <ul style="list-style-type: none"> ◊ Die anderen Bestandteile können variieren: ◊ Zugriffsrechte dürfen in abgeleiteter Klasse erweitert sein ◊ <code>private</code> → <code>ε</code> → <code>protected</code> → <code>public</code> ◊ Bei Referenztypen Rückgabetyt durch Subtyp ersetzbar ◊ Exceptionklassen durch Subtypen ersetzbar ▷ Aufruf der überschriebenen Methode mit <code>super.m()</code> ; ▷ Exceptions: <ul style="list-style-type: none"> ◊ Exception Klasse darf durch Subtyp ersetzt werden
Overload	▷ Methoden mit selbem Bezeichner, aber unterschiedlicher Parameterliste ▷ Die Methode wird überladen ▷ Konstruktoren kann man auch überladen <ul style="list-style-type: none"> ◊ Für manche Werte werden dann Standardwerte gesetzt ◊ Anderer Konstruktor auch in Konstruktor aufrufbar (<code>this(1);</code>) ▷ Alle Methoden einer Klasse müssen unterschiedliche Signatur haben
Subtypen	▷ Abgeleitete Klassen / Interfaces (extends) ▷ Überall wo ein Referenztyp (Supertyp) erwartet wird: <ul style="list-style-type: none"> ◊ Verwendung eines Objekts eines Subtyps möglich <ul style="list-style-type: none"> in Zuweisung an Variable als Parameterwert als Rückgabewert
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 ▷ Entscheidet , auf welche Attribute/Methoden zugegriffen werden darf <ul style="list-style-type: none"> ◊ Müssen im statischen Typ vorhanden sein (definiert oder ererbt)
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 ▷ Entscheidet , welche Implementation der Methode aufgerufen wird
Downcast	▷ <code>if (y instanceof X) {...}</code> <ul style="list-style-type: none"> ◊ 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 ▷ Downcast <ul style="list-style-type: none"> ◊ Vorherige Überprüfung mit <code>isinstanceof</code> ◊ Ermöglicht z.B.: <code>X z;</code> <code>z = (X) y;</code> ◊ Warum? Zugriff auf Funktionen, die nicht im statischen Typ existieren
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"> ▷ byte 8 Bits ▷ short 16 Bits ▷ int 32 Bits ▷ long 64 Bits
Binärdarstellung	<ul style="list-style-type: none"> ▷ Nicht-negative Zahlen: <ul style="list-style-type: none"> ◊ Führendes Bit = null ▷ Negative Zahlen: <ul style="list-style-type: none"> ◊ Führendes Bit = eins ▷ Führendes Bit auch Vorzeichenbit genannt ▷ Größte darstellbare Zahl: $2^{N-1} - 1$ // Jedes Bit außer dem Ersten gesetzt
Umwandlung nicht-negativ in Dezimal	<ul style="list-style-type: none"> ▷ Veranschaulichung am Datentyp byte <ul style="list-style-type: none"> ◊ byte maximal 127, deswegen reichen Zehnerpotenzen bis 100 ▷ Vorhergehensweise: (Beispiel 01101101 / 109) <ul style="list-style-type: none"> ◊ $01101101_2 / 01100100_2 = 00000001_2 \rightarrow "1?"$ <ul style="list-style-type: none"> - Ganzzahlige Division der Zahl durch 100 \rightarrow 1 Rest 9 ◊ $01101101_2 \% 01100100_2 = 00001001_2$ <ul style="list-style-type: none"> - Rest der vorherigen Division \rightarrow 9 ◊ $00001001_2 / 00001010_2 = 00000000_2 \rightarrow "10?"$ <ul style="list-style-type: none"> - Teilen durch 2.höchste Zehnerpotenz (10^1) \rightarrow 0 ◊ $00001001_2 \% 00001010_2 = 00001010_2$ <ul style="list-style-type: none"> - Rest der vorherigen Division \rightarrow 9 ◊ $00001001_2 / 00000001_2 = 00001001_2 \rightarrow "109"$ <ul style="list-style-type: none"> - Teilen durch kleinste Zehnerpotenz (10^0) \rightarrow 9 - Teilen durch 1 natürlich überflüssig
Unicode-Kodierung	<ul style="list-style-type: none"> ▷ Darstellung der Dezimalziffern: <ul style="list-style-type: none"> ◊ Setzen des 16er und 32er Bits auf 1 ◊ Also Addieren von 48 ◊ Bereich der Zahlenwerte: 48 – 57
Umwandlung Dezimal in Datentyp	<ul style="list-style-type: none"> ▷ Jede Zeichen der Zahl (z.B. 3856) stellt ja eine char-Zahl dar <ul style="list-style-type: none"> ◊ Subtrahieren von 48 (siehe Unicode Kodierung) ◊ Multiplikation mit dazugehöriger Zehnerpotenz ◊ Addieren der einzelnen Werte ▷ Schleife über Zehnerpotenzen, so oft wie die Zahl Ziffern enthält
Negative Zahlen (Zweierkomplement)	<ul style="list-style-type: none"> ▷ Umwandlung von positiv nach negativ: <ul style="list-style-type: none"> ◊ Binärdarstellung der positiven Zahl ◊ Umdrehen aller Bits (1-Komplement) ◊ Addieren einer 1 (2-Komplement) ◊ Auch rückwärts anwendbar ▷ Zweierkomplement ermöglicht einfache Darstellung der 0 (0000) ▷ Zweierkomplement ermöglicht einfache Subtraktion: <ul style="list-style-type: none"> ◊ Setzen des Subtrahenden ins Zweierkomplement ◊ Addieren der beide Werte ▷ Negativer Bereich ist um eins größer als Positiver

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

Gebrochene Zahlen

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

28 Anhang: Korrekte Software

Korrektheit auf einzelnen Abstraktionsebenen

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

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

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

29 Anhang: Effizienz von Software

Nebenaspekte der Effizienz

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

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

Fehlervermeidender Entwurf

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

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

Abstraktion und Polymorphie

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

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

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**