

FOP Reference Sheet

Jonas Milkovits

Last Edited: 7. April 2020

Inhaltsverzeichnis

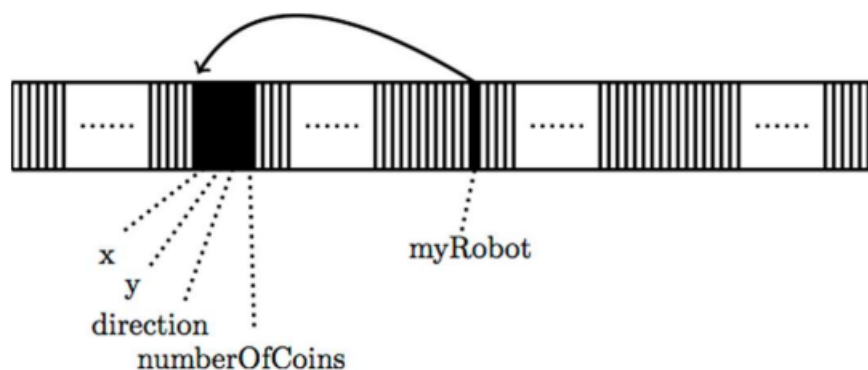
1	Stuff that I skipped cuz of chapter 4	1
2	Computerspeicher	1
3	Datenstrukturen	2
4	Datentypen	2
5	Exceptions (java.lang.Exception;)	3
6	Fehler	4
7	Graphics (java.awt.Graphics;)	4
8	Interfaces	4
9	JUnit-Tests	5
10	Klassen	6
11	Konversionen	6
12	Methoden	7
13	Packages und Zugriffsrechte	8
14	Programme und Prozesse	8
15	Schleifen, if, switch	9
16	String (java.lang.String)	9
17	Syntax	10
18	Vererbung	11

1 Stuff that I skipped cuz of chapter 4

Exceptions aus Lambda-Ausdrücken	▷ Kapitel 5: 47 - 50
	▷
	▷
	▷
	▷
	▷
	▷
	▷

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 Methodenaufwurf 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 ▷ Methodenaufwurf 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	<ul style="list-style-type: none"> ▷ 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 char	<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-ressources</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 Throwable 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 ◊ 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

6 Fehler

Kompilierzeitfehler (compile-time errors)	▷ Falsche Klammersetzung, falsche Schlüsselwörter,.. ▷ Programm wird nicht übersetzt ⇒ Fehlermeldung vom Compiler
Laufzeitfehler (run-time errors)	▷ Tritt während der Ausführung auf ▷ Führt zum Abbruch des Programms ⇒ Ausgabe der Fehlermeldung ▷ Kann nicht vom Compiler entdeckt werden ▷ IndexOutOfBounds, NullPointerException,..

7 Graphics (java.awt.Graphics;)

Applet	▷ 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> <code>public void paint (Graphics graphics) {...}</code> Klasse <code>Graphics</code> verknüpft Programm mit Zeichenfläche ▷ 2.1 <code>GeomShape2D</code> -Array <code>GeomShape2D pic = new GeomShape2D[3];</code> Füllen des erstellten Arrays mit Formen (z.B.: <code>new Circle(0,0,0);</code>) ▷ 2.2 Erstellen jeder Form mithilfe Randfarbe, Füllfarbe und Zeichnen <code>pic[0].setBoundaryColor(Color.RED); // Randfarbe</code> <code>pic[0].setFillColor(Color.RED); // Füllfarbe</code> <code>pic[0].paint(graphics); // Eigentliches Zeichnen</code>
GeomShape2D	▷ Abstrakte Klasse (Methode <code>paint</code> ist abstrakt) ▷ Attribute: <code>int positionX; int positionY; int rotationAngle;</code> <code>int transparencyValue; Color boundaryColor; Color fillColor;</code> ▷ Subklassen: <code>Rectangle</code> , <code>Circle</code> , <code>StraightLine</code>

8 Interfaces

Erzeugung	▷ Meist in eigener Datei ▷ <code>public interface MyInterface {...}</code> ▷ Alle Methodes und das Interface müssen <code>public</code> sein
Methoden	▷ 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
Verwendung	▷ <code>implements MyInterface</code> nach Klassenname ▷ Beliebig viele Interfaces möglich (seperiert durch ,) ▷ Ein Interface kann mehrere andere Interfaces erweitern (<code>extends</code>

9 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▷ <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 <code>java.lang.reflect</code>
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

10 Klassen

Erzeugung	<ul style="list-style-type: none"> ▷ meist in seperater .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)

11 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>

12 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.: <code>public</code>) ▷ 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 (1.42, 2.7) ◊ 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

13 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: <code>package mypackage;</code><ul style="list-style-type: none">◊ Datei gehört damit zum Package <code>mypackage</code>◊ <code>mypackage</code> wird automatisch importiert
Import	<ul style="list-style-type: none">▷ <code>import package.*;</code>▷ <code>*</code> steht für alle Definitionen aus <code>package</code>▷ <code>*</code> 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: <code>java.lang.*</code> (automatisch importiert)▷ Konstanten: <code>import static java.lang.Math.PI;</code><ul style="list-style-type: none">◊ Ermöglicht Schreiben von <code>PI</code> statt <code>Math.PI</code>
Zugriffsrechte	<ul style="list-style-type: none">▷ Klassen/Enum: nur <code>public</code> oder nichts<ul style="list-style-type: none">◊ Nur eine Klasse darf <code>public</code> sein (Damit auch Dateiname)▷ <code>private</code>: Zugriff innerhalb der Klasse▷ Keine Angabe: <code>private</code> + im Package▷ <code>protected</code>: Keine Angabe + in allen Subklassen▷ <code>public</code>: <code>protected</code> + an jeder Import-Stelle

14 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	<ul style="list-style-type: none">▷ CPU besteht aus mehreren Prozessorkernen▷ Mehrere Prozesse laufen dementsprechend parallel▷ Allerdings bearbeitet jeder Kern nur einen Prozess gleichzeitig (sehr kurz)<ul style="list-style-type: none">◊ Illusion von Multitasking

15 Schleifen, if, switch

while-Schleife	<ul style="list-style-type: none">▷ <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	<ul style="list-style-type: none">▷ <code>do {Anweisung;} while (Bedingung);</code>▷ Anweisungsblock wird immer mindestens einmal ausgeführt
for-Schleife	<ul style="list-style-type: none">▷ <code>for (Anweisung davor; Bedingung; Anweisung danach) {Anweisung}</code>▷ z.B.: <code>for (int i = 0; i < 10; i++) {...}</code><ul style="list-style-type: none">◊ Zehnmalige Ausführung der Anweisung▷ Kurzform: <code>for (Position p : positions) {}</code><ul style="list-style-type: none">◊ (Komponententyp Identifier : ArrayName)
if-Anweisung	<ul style="list-style-type: none">▷ <code>if (Bedingung) {...}</code><ul style="list-style-type: none">◊ Führt den Code in der Anweisung nur aus, falls die Bedingung erfüllt ist▷ <code>if (Bedingung) {} else {}</code><ul style="list-style-type: none">◊ Code, der ausgeführt wird, falls Bedingung nicht erfüllt ist
switch-Anweisung	<ul style="list-style-type: none">▷ Abfrage von mehreren Fällen▷ <code>switch (i) { case 2: ... break; case 3: ... break; default: ... }</code>▷ <code>break</code>; Ohne <code>break</code>, 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

16 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

17 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>,...
Identifier	<ul style="list-style-type: none">▷ Namen für Klassen, Variablen, Methoden,..▷ Erstes Zeichen darf keine Ziffer sein▷ Keine Keywords als Identifier▷ Identifier 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
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>

18 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 Rückgabotyp 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