FOP Reference Sheet

Jonas Milkovits

Last Edited: 12. April 2020

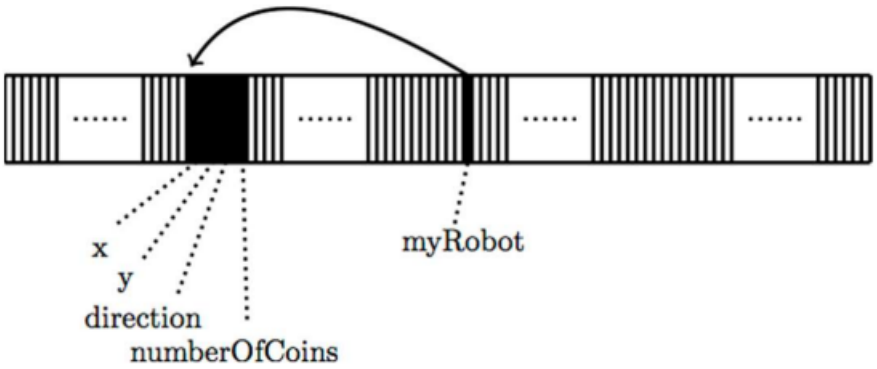
Inhaltsverzeichnis

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 abgespeichet 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: `int[] test = new int[n];` ▷ `n` gibt in diesem Fall die feste Anzahl der speicherbaren Variablen an ▷ Natürlich auch Arrays von Objekten möglich ▷ Zugriff auf Variablen: `test[0]` für ersten Wert (Index) ▷ Zugriff auf Länge: `test.length`

4 Datentypen

Konstanten	▷ Variable/Referenz wird dadurch unveränderbar ▷ z.B.: `final myClass ABC = new myClass();` ◇ Referenz zwar nicht veränderbar, Objekt aber schon ▷ `Integer.MAX_VALUE / Integer.MIN_VALUE` ▷ Unendlich: `Double.POSITIVE_INFINITY / Double.NEGATIVE_INFINITY` ▷ Müssen initalisiert werden
Primitive Dateitypen	▷ Ganze Zahlen: byte → short → int → long ▷ Gebrochene Zahlen: float → double ▷ Logik: boolean ▷ Zeichen: char ▷ Mehrere Definitonen: `int m = 1, n, k = 2;` ▷ Ohne Initialisierung: undefinierter Wert
Literale	▷ wörtlich hingeschriebene Werte eines Datentyps ▷ Zahlen standardmäSSig int, falls `long` gewünscht: `123L oder 123l` ▷ Bei gebrochenen double, falls `float` gewünscht: `12.3F oder 12.3f` ▷ null: Nutzung für Referenzen → verweist auf nichts
Boolean	▷ nur `true` und `false` ▷ Negation `!a` ▷ Logisches Und: `a && b` ▷ Logisches Oder: `a || b` (inklusiv) ▷ Gleichheit: `a == b`
Zeichentyp char	▷ z.B.: `char c = 'a';` ▷ Interne Kodierung als Unicode ▷ \t Horizontaler Tab ▷ \b Backspace ▷ \n Neue Zeile ▷ Auch Darstellung im Hexacode (\u039A)

Enumeration	▷ Zusammenfassung mehrerer Konstanten (feste Anzahl)
	▷ Erzeugung meist in eigener .java Datei
	▷ `enum MyDirection {DOWN, RIGHT}`
	▷ Keine Objekterzeugung von Enumeration möglich
	▷ Abspeichern in Variable des Enum-Types ist jedoch möglich
	▷ `MyDirection dir = MyDirection.DOWN;`
	▷ Klassenmethoden:
	◇ `values() // Returns array with all enum components`
	◇ `name() // Returns the name of the calling object as string`

Referenztypen	▷ Alle Typen, die keine primitiven Datentypen sind ▷ Unterscheidung zwischen Referez und eigentlichem Objekt ▷ Gleichheitsoperator `==` vergleicht nur die Referenz (Objektidentität) ◇ Verweis auf dasselbe Objekt ▷ Wertgleichheit bezieht sich auf das Objekt an sich ◇ Deep Copy ⇒ An allen parallelen Stellen Wertgleichheit ◇ Shallow Copy ⇒ Nur Kopie der Adressen ▷ Ohne Initialisierung: Null

5 Exceptions (java.lang.Exception;)

Exception-Klassen	▷ Alle Klassen, die direkt oder indirekt von java.lang.Exception abgeleitet sind ▷
Exception werfen	▷ `throws Exception {...}` nach Parameterliste im Methodenkopf ▷ Dies signalisiert, dass die Methode mindestens einen Fehler wirft ▷ Die geworfene Exception muss vom `throws`-Typ oder Subtyp sein ▷ Auch mehrere Exceptions möglich, mit einem Komma getrennt ▷ Werfen der Exception: ◇ z.B.: `throw new Exception (No lower case letter!");` ◇ Hier wird als Parameter für die Objekterstellung ein String übergeben ▷ `throws`: ◇ Führt zur Beendung der Methode ◇ Liefert das geworfene Exception-Objekt zurück

Exception fangen	▷ Bei Methoden, die Exceptions werfen, wird ein `try-catch`-Block benötigt
	▷ Aufbau:
	◇ Methoden, die Exceptions werfen in `try {...}` aufrufen
	◇ Falls Exception auftritt wird `catch (Exception exc) {...}` aufgerufen
	◇ `catch` muss direkt im Anschluss nach `try` stehen
	◇ Falls kein Fehler auftritt, wird `catch` übersprungen
	◇ Das Programm wird dann normal weiter ausgeführt
	▷ Es sind auch mehrere `catch`-Blöcke mit versch. Parametern möglich
	▷ Methoden:
	◇ `getMessage(); // Returns the error message as a string`
	◇ `printStackTrace(); // Ausgabe des Call-Stacks`
	▷ Alle möglichen Exceptions müssen durch den `catch`-Block abgedeckt sein
	▷ Falls Exception zu mehreren `catch`-Blöcken 'passt', wird der Erste ausgeführt
	◇ Deswegen Reihung der `catch`-Blöcke von Subtyp nach Supertyp
	▷ Auch mehrere Exceptions in einem `catch`-Block möglich mit `||`

Weiterreichen	▷ Weiterreichen der Fehlermeldung durch `throws` im Methodenkopf möglich ▷ Kein `try-catch`-Block notwendig ▷ Main-Methode kann z.B. keine Exceptions weiterreichen
`try-with-ressources`	▷ Für Ressourcen, die unbedingt wieder geschlossen werden müssen ▷ Öffnung der Ressource in runden Klammern: `try (Printer p =... ) {...}` ▷ Mehrere Ressourcen möglich, getrennt durch Semikolon
Runtime Exceptions	▷ Ausnahme zu `try`-Blöcken ▷ Exceptions von java.lang.RuntimeException und Subtypen ▷ z.B.: IndexOutOfBoundsException, NullPointerException ▷ Grund: Vermeidung von dauerenden `try`-Blöcken
Throwable und Error	▷ Exception und Error sind beide von Throwable abgeleitet ▷ Alle drei befinden sich im Paket java.lang ▷ Error: ◇ Werden geworfen, falls Fehlerbehandlung keinen Sinn macht ◇ Programmabbruch als Ausweg ▷ AssertionError: ◇ `throw new AssertionError("Bad!");` ◇ Kurzform: `assert x == 2: "Bad!";` ◇ **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 Generics

Wrapper-Klassen	▷ 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:
	◇ `intValue(); // Returns specific value of class`
	◇ `MAX_VALUE; // Returns max value`
	▷ Boxing/Unboxing:
	◇ Primitiver Datentyp und Wrapper-Klasse sind austauschbar
	◇ Automatische Umwandlung ineinander
	◇ Boxing: `Integer i = 123;`
	◇ Unboxing: `System.out.print(i); // 123`

Generische Klassen	▷ `public class Pair <T1, T2> {...}` ▷ 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 Typparamter festgelegt ◇ `Pair<Integer,Double> pair = new Pair<Integer,Double>(2,3.5);` ◇ `Pair` ist mit `Integer` und `Double` **instanziiert** ◇ Typparameter können natürlich auch vom selben Typ sein

Generische Methoden	▷ Auch in **nicht-generischen Klassen** generische Methoden möglich
	▷ `public class X {...}`
	▷ Einzelne Methode parametrisiert:
	◇ `public <T1,T2> Pair<T1,T2> makePair(T1 t1, T2 t2) {...}`
	◇ **Parametrisierung** der Methode (`<T1,T2>`) steht vor dem Rückgabetyp
	▷ Aufurf:
	◇ `Pair<A,B> pair1 = x.makePair(new A(), new B());`
	◇ Compiler erkennt selbst die Typen für die Methode
	▷ Falls `T1` z.B. schon die Klasse `X` parametrisiert:
	`public class X <T1> {`
	`public <T2> Pair<T1,T2> makePair(T1 t1, T2 t2) {...}`
	`}`

Typparameter	▷ Alle Arten von Klassen und Arrays möglich ▷ Auch parametrisierte Klassen sind als Typparamter möglich ▷ Typparameter dürfen jedoch nicht vom primitiven Datentyp sein ▷ Vererbung von Typparametern ist jedoch nicht übertragbar ◇ Bei bereits instanziierten Parametern sind keine Subklassen möglich ▷ Kurzform: ◇ `Pair<String,Integer> pair;` ◇ `pair = new Pair<> ("Hello", 123);` ◇ `"Diamond-Operator"`: Compiler erkennt selbstständig die Instanziierung

Eingeschränkte Typparameter	▷ Werden bei der Definition von generischen Klassen/Methoden verwendet ▷ `<T extends X> // T gleich X, oder direkt/indirekt Subtyp von X` ◇ Notwendig um sicherzustellen, dass aufgerufene Methoden definiert sind ◇ z.B.: `<T extends Number> // Methoden wie doubleValue() immer vorhanden` ▷ Mehrfache Einschränkung: ◇ `<T extends X & Interface1 & Interface2` ◇ Klasse muss, falls vorhanden, an erster Stelle stehen
Wildcards	▷ Werden bei der Instanziierung von Typparametern verwendet ▷ `public double m (X<? extends Number> n) {...}` ◇ Ermöglicht nun die Verwendung von Subklassen bei aktualen Parametern ◇ (Siehe Einschränkung `Typparameter / 4. Stichpunkt`) ▷ **Unterschied:** ◇ `public <T extends Number> double m (X<T> n) {...}` ◇ Generische Methode mit eingeschränkt wählbarem Typparameter ◇ `public double m (X<? extends Number> n) {...}` ◇ Nichtgenerische Methode mit generischem Parameter mit eingeschränkt wählbarem Typparameter ▷ Weitere Wildcard: `X<?>` ◇ Allgemeinst möglichste, `extends Object` ▷ `X<? super Double>` ◇ Mit allen Supertypen (direkt/indirekt) und alle implementieren Interfaces
Empfehlungen	▷ Oracle-Empfehlungen im Bezug auf Wildcards ▷ In-Parameter (Werte einer Methode, die nur gelesen werden): ◇ Verwendung von `extends` ▷ Out-Paramter (Werte einer Methode, die nur geschrieben werden): ◇ Verwendung von `super` ▷ In/Out-Parameter: ◇ Keine Verwendung von Wildcards ▷ Rückgaben: ◇ Keine Verwendung von Wildcards
Interface Comparator	▷ `Functional Interface` im Package `java.util` ▷ Verwendung: ◇ Erstellen einer Vergleichsklasse, die `Comparator<T>` implementiert ◇ `..class MyComp<T extends Number> implements Comparator<T> {...}` ◇ Generisch mit einem Typparameter ▷ Methode: `public int compare (T t1, T2) {...}` ◇ 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 `compareTo`: sortiert lexikographisch
Einschränkungen	▷ Keine primitiven Datentypen als Instanziierung von Typparametern ▷ Keine Erzeugung von Objekten/Arrays von Typparametern mit `new` ▷ Keine Klassenattribute von Typparametern ▷ Kein Downcast oder `instanceof` von Typparametern ▷ Kein `throw-catch` mit Typparametern ▷ Keine Methodenüberladung mit Typparametern

8 Graphics (java.awt.Graphics;)

Applet	▷ leichtgewichtige Variante an Graphikprogrammen
	▷ `import java.awt.Applet;`
	▷ 1. Erstellen eigener Applet-Klasse (`extends Applet`)
	▷ 2. Überschreiben der Methode `paint`
	`public void paint (Graphics graphics) {...}`
	Klasse `Graphics` verknüpft Programm mit Zeichenfläche
	▷ 2.1 `GeomShape2D`-Array
	`GeomShape2D pic = new GeomShape2D[3];`
	Füllen des erstellten Arrays mit Formen (z.B.: `new Circle(0,0,0);`)
	▷ 2.2 Erstellen jeder Form mithilfe Randfarbe, Füllfarbe und Zeichnen
	`pic[0].setBoundaryColor(Color.RED); // Randfarbe`
	`pic[0].setFillColor(Color.RED); // Füllfarbe`
	`pic[0].paint(graphics); // Eigentliches Zeichnen`

GeomShape2D	▷ Abstrake Klasse (Methode `paint` ist abstrakt)
	▷ Attribute:
	`int positionX; int positionY; int rotationAngle;`
	`int transparencyValue; Color boundaryColor; Color fillColor;`
	▷ Subklassen: `Rectangle, Circle, StraightLine`

9 Interfaces

Erzeugung	▷ Meist in eigener Datei ▷ `public interface MyInterface {...}` ▷ Alle Methodes und das Interface **müssen** `public` sein
Methoden	▷ Werden hier nicht implementiert, sondern nur definiert ▷ `public` kann weggelassen werden, da ohnehin notwending ▷ Implementierte Methoden müssen dann auch `public` sein ▷ Falls eine der Methoden nicht implementiert wird ⇒ Klasse abstrakt
Verwendung	▷ `implements MyInterface` nach Klassenname ▷ Beliebig viele Interfaces möglich (seperiert durch ,) ▷ Ein Interface kann mehrere andere Interfaces erweitern (`extends`

10 JUnit-Tests

Allgemein	▷ Tests als Ganzes - Black-Box-Tests ▷ JUnit-Tests werden in eine seperate Quelldatei geschrieben ▷ Die zu testende Einheit/Klasse wird dann importiert
Imports	▷ `import static org.junit.Assert.assertEquals;` ▷ `import static org.junit.Assert.asserTrue;` ▷ `import org.junit.jupiter.api.Test;` ▷ `import org.junit.jupiter.api.BeforeEach;` ▷ `import static org.junit.jupiter.api.Assertions.assertThrows;`

Methoden:	▷ `assertEquals(...,...); // true, falls beide Parameter identisch`
	◇ Existiert auch mit 3 Parametern, 3. Wert entspricht maximalen Unterschied
	▷ `assertTrue(...); // true, falls der Parameter true ist`
	▷ `assertThrows(...,...); // Wirft Exception abhängig von Executable`
	◇ Erster Parameter zu werfende Exception.class
	◇ Zweiter Paramter Functional Interface aus dem Package java.lang.reflect

Test	▷ `@Test` vor der Methode ▷ `void` als Rückgabewert ▷ Nutzung einer `assert`-Methode (siehe Methoden)
BeforeEach	▷ `@BeforeEach` vor der Methode ▷ Wird vor jeder einzelnen Testmethode einmal ausgeführt

11 Klassen

Erzeugung	▷ meist in seperater .java Datei ▷ `public class MyClass {}` ▷ `new MyClass();`: ◇ Reserviert ausreichend Speicherplatz für das Objekt ▷ `MyClass x = new MyClass();`: ◇ Speichern der Adresse des neuen Objekts in der Referenz x
Attribute	▷ Eigenschaften der Objekte/Klassen ▷ z.B.: `private int x;` (Objektattribut) ▷ z.B.: `private static int x;` (Klassenattribut)
Konstruktor	▷ Wird zur Erzeugung von neuen Objekten einer Klasse verwendet ▷ Methode mit selben Namen wie Klasse und ohne Rückgabetyp ▷ z.B.: `public MyClass (int x, int y) {this.x = x; this.y = y;}` ▷ Erzeugung eines neuen Objekts: `MyClass test = new MyClass(2,4);` ▷ Falls kein Konstruktur angegeben wird → Default Constructor ◇ Basisklasse muss auch Konstruktor mit leerer Parameterliste haben ▷ Konstruktoren werden `nicht` vererbt ▷ `Static Initializer` ◇ Methodenkopf besteht nur aus `static {...}` ◇ Wird genutzt um auf jeden Fall Klassenkonstanten zu initialisieren ▷ Aufruf anderen Konstruktors in Konstruktor mit `this(Parameter);`
Abstraktion	▷ `abstract public class MyClass {...}` ▷ Notwendig, sobald Klasse eine abstrakte Methode beinhaltet ▷ Keine Objekterzeugung möglich ▷ Meist als Klasse mit Rahmenbedingungen für Subklassen verwendet

Klasse aller Klassen	▷ `java.lang.Object`
	▷ Jede Klasse ist direkt oder indirekt von `Object` abgeleitet
	▷ Methoden:
	◇ `boolean equals (Object obj) {...} // Test auf Wertgleichheit`
	◇ `String toString() {...} // Zustand des Objekts als String`
	◇ Werden oft an jeweilige Klasse angepasst

Verborgene Informationen	▷ Jedes Objekt einer Klasse erhält einen Verweis auf ein anonymes Objekt ▷ Dieses anonyme Objket wird für jede Klasse nur einmal eingerichtet ▷ Enthät Informatiuonen zur Klasse, Attribute und Methoden der Klasse ▷ Methodentabelle: ◇ Gibt an, welche Implementationen aller Methoden verwendet wird ◇ Ermöglicht, die Feststellung der Klasse zur Laufzeit ◇ Methode in Supertyp und Substyp haben den selben Index (Position)

12 Konversionen

Implizit	▷ Immer möglich, wenn kein Informationsverlust entstehen kann ▷ z.B.: kleinerer Datentyp in gröSSeren
Explizit	▷ Meist Informationsverlust ▷ Durchführung durch Angabe des Datentyps in Klammern davor ▷ z.B.: `int i = (int)testDouble;`

13 Methoden

Methodenaufbau	▷ Modifier Rückgabewert Identifier (Parameterliste) {Anweisung} ▷ Alles vor den Anweisung: Methodenkopf (Head) ▷ Alles in den geschweiften Klammern: Methodenrumpf (Body) ▷ z.B.: `public void setX (int x) {this.x = x;}` (Objektmethode) ▷ z.B.: `public static void setY (int y) {this.y = y;}` (Klassenmethode) ▷ `this.x` steht hier für das Objektattribut und nicht den Parameter
Ausführung	▷ Objektmethoden: `myObject.setX(2);` ▷ Klassenmethoden: `MyClass.setY(2);`
return	▷ Wird für Rückgabe bei Methoden mit Rückgabewert benötigt
Abstraktion	▷ `abstract` vor Modifier (`z.B.: public`) ▷ Abstrakte Methoden haben keinen Methodenrumpf
Parameter	▷ Parameterliste in Definition: Formale Parameter ▷ Parameterliste bei Methodenaufruf: Aktuale Parameter ◇ Kommt von actual ⇒ tatsächlich, vorliegend ▷ Verhalten bei Referenzen: ◇ Kopie der Adresse des Objekts bei Initialisierung des formalen durch aktualen Parameter ▷ Variable Parameterzahl: ◇ `void m (double... args) {...}` ◇ 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	▷ Besteht aus Identifier und Parameterliste ▷ Eine Klasse kann keine zwei Methoden mit derselben Signatur haben
Klassenmethoden	▷ 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

14 Packages und Zugriffsrechte

Package	▷ 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	▷ `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	▷ 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

15 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

16 Schleifen, if, switch

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

17 String (java.lang.String)

Eigenschaften	▷ Sonderrolle, da Klasse, aber trotzdem Literale in Java ▷ Zeichenketten, die aus allen möglichen chars bestehen

Methoden:	▷ `String str = "Hello World";`
	◇ `str.length; // 11`
	◇ `str.charAt(2); // e`
	◇ `str.indexOf('e'); // 2`
	◇ `str.matches("He.+rld"); // true`
	`.+` ⇒ `.` als Platzhalter für beliebiges Zeichen, `+` erlaubt Wiederholung
	⇒ Regular Expression
	◇ `String str 2 = str.concat("b"); // Anhängen`
	◇ `String str 2 = str1 + "b"; // Kurzform`

18 Syntax

Keywords	▷ Können nur an bestimmten Stellen im Code stehen ▷ z.B. `class, import, public, while,..`
Identifier	▷ Namen für Klassen, Variablen, Methoden,.. ▷ Erstes Zeichen darf keine Ziffer sein ▷ Keine Keywords als Identifier ▷ Identifier sind case-sensitive
Konventionen	▷ Variablen / Methoden beginnen mit Kleinbuchstaben (`testInt`) ▷ Klassen beginnen mit GroSSbuchstaben (`testClass`) ▷ Wortanfänge im Inneren mit GroSSbuchstaben ▷ Konstanten bestehen aus _ und GroSSbuchstaben (`CENTS_PER_EURO`) ▷ Packagenamen nur aus Kleinbuchstaben und _ bei unzulässigen Zeichen
Kommentare	▷ `//` Einzelne Zeile ▷ `/*...*/` Mehrere Zeilen ▷ `/**...*/` Erzeugung von Javadoc
Javadoc	▷ Erzeugung mithilfe von `/**` und Enter ▷ Bei Methodenköpfen: ◇ `@param x the dividend` ◇ `@return x divided by x` ◇ `@throws class IndexOutOfBoundsException if c is not an int` ▷ Bei Quelldateien: ◇ `@author` ◇ `@version`
Rechtsausdrücke	▷ Haben Typ und Wert ▷ z.B.: `2*3+1`
Linksausdrücke	▷ Verweisen auf Speicherstellen ▷ z.B.: `int n`

19 Vererbung

Zweck	▷ Weitergabe von allen Methoden und Attributen
Verwendung	▷ `public class MySubClass extends MyClass {}`
Konstruktor	▷ Aufruf des Konstruktors der Superklasse mithilfe von `super(Parameter);` ▷ Dieser Aufruf erfolgt im Konstruktor der Subklasse ▷ z.B.: `public MySubClass (int x) { super(x);<v}`
Overwrite	▷ Methoden in Subklassen können auch neu geschrieben werden ◇ Die Implementation der Superklasse wird sozusagen überschrieben ▷ Selber Name und Parameterliste notwendig ▷ Signatur der Methoden muss identisch sein ◇ Die anderen Bestandteile können variieren: ◇ Zugriffsrechte dürfen in abgeleiteter Klasse erweitert sein ◇ private $\rightarrow \epsilon \rightarrow$ protected $\rightarrow$ public ◇ Bei Referenztypen Rückgabetyp durch Subtyp ersetzbar ◇ Exceptionklassen durch Subtypen ersetzbar ▷ Aufruf der überschriebenen Methode mit `super.m();` ▷ Exceptions: ◇ Exception Klasse darf durch Subtyp ersetzt werden
Overload	▷ Methoden mit selbem Bezeichner, aber unterschiedlicher Parameterliste ▷ Die Methode wird überladen ▷ Konstruktoren kann man auch überladen ◇ Für manche Werte werden dann Standardwerte gesetzt ◇ Anderer Konstruktor auch in Konstruktor aufrufbar (`this(1);`) ▷ Alle Methoden einer Klasse müssen unterschiedliche Signatur haben
Subtypen	▷ Abgeleitete Klassen / Interfaces (`extends`) ▷ Überall wo ein Referenztyp (Supertyp) erwartet wird: ◇ Verwendung eines Objekts eines Subtyps möglich 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.: `X a = new Y();` ⇒ `X` hier statischer Typ ▷ **Entscheidet**, auf welche Attribute/Methoden zugegriffen werden darf ◇ 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.: `X a = new Y();` ⇒ `Y` hier dynamischer Typ ▷ **Entscheidet**, welche Implementation der Methode aufgerufen wird
Downcast	▷ `if (y instanceof X) {...}` ◇ Gibt `true` zurück, falls `y` (Variable von Referenztyp) gleich dem Typen von `X` oder ein Subtyp von `X` ist ▷ Downcast ◇ Vorherige Überprüfung mit `isinstanceof` ◇ Ermöglicht z.B.: `X z;` `z = (X) y;` ◇ **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