FOP Reference Sheet

Jonas Milkovits

Last Edited: 13. April 2020

Inhaltsverzeichnis

1 Stuff that I skipped cuz of chapter 4

Exceptions aus Lambda-Ausdrücken	▷ Kapitel 5: 47 - 50
Listen von Lambda-Ausdrücken	▷ Kapitel 7: 60 - 65
Methodennamen als Lambda-Ausdrücke	▷ Kapitel 8: 55 - 84
Streams in Racket	▷ Kapitel 8: 122 - 133

2 Collections

Informationen	▷ Sammlungen von Elementen (Objekte eines generischen Typs) ▷ Struktur: ◇ Alle Klassen und Interfaces in `java.util` ◇ Interface `Collection`: Alle Klassen implementieren dieses Interface ◇ Klasse `Collections`: Basisalgorithmen, Sortieren ◇ Interface `List`: Erweitert `Collection`, mehr Funktionalitäten ◇ Klasse `Iterator`: Iteration über die Elemente einer `Collection` ▷ Beispiele für Klasse, die das Interface `Collection` implementieren: ◇ `Vector, LinkedList, ArrayList, TreeSet, HashSet`
Interface `Collection`	▷ z.B.: `Collection<Number> c1 = new ArrayList<Number>();` ◇ Speichert leere `ArrayList` in einer Referenz des Interface `Collection` ◇ Dies ist möglich, da `ArrayList` das Interface `Collection` implementiert ▷ Methoden: ◇ `add` - Fügt zur ArrayList ein neues Element hinzu - Gibt `true` zurück, falls Hinzufügen erfolgreich ◇ `addAll` - Hat eine `Collection` als Parameter und fügt diese hinzu ◇ `size` - Anzahl der Elemente als int ◇ `isEmpty` - `true`, falls `Collection` keine Elemente enthält (`size == 0`) ◇ `contains` - Parameter vom Typ `Object` - Überprüft, ob aktualer Parameter in `Collection` vorhanden ist - Nutzt `equals` von `Object` → Wertgleichheit ◇ `containsAll` - `true`, falls ganze übergebene `Collection` enthalten ist ◇ `clear` - Entfernt alle Elemente aus der `Collection` ◇ `remove` - Entfernt übergebenes `Object` - `true`, falls `Object` mindestens einmal vorhanden - Bei mehreren, entscheidet die `Collection`-Klasse welches entfernt wird

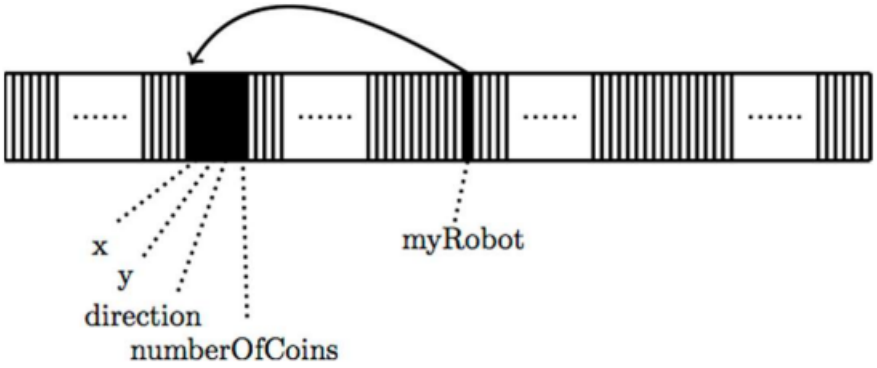
Interface `List`	▷ Erweitert das Interface `Collection` ▷ Unterschied: Definition einer Reihenfolge auf den Elementen ▷ Methoden: ◇ `indexOf` - Liefert ersten Index zurück, an dem `Object` zu finden ist - Liefert -1 zurück, falls Parameter nicht in Liste gefunden wird ◇ `set` - `T set(int index, T element) ...` - Ersetzt Element an Stelle `index` durch `element` - Gibt ersetztes Element zurück ◇ `add` - Identisch zu Methode `set`, jedoch ein Unterschied: - Überschreibt das Element **nicht**, sondern fügt es vor dem Element ein
Sortieren mit Comparator	▷ Klasse `Collections` hat Klassenmethode `sort` ▷ `Collections.sort(list, new MyComparator());` ◇ Erster Parameter: Zu sortierende Liste (z.B.: `List<Student> list = ...`) ◇ Zweiter Parameter: Selbst erstellte Sortierlogik ◇ Typparameter von `Comparator` und `List` müssen gleich sein
Interface `Iterator`	▷ `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` ▷ `Collection<Number> c1 = new ArrayList<Number>();` ▷ `Iterator<Number> it1 = c1.iterator();` ◇ `Collection` besitzt die Methode `iterator()` ◇ Liefert ein Objekt ihrer eigenen `Iterator`-Klasse zurück ▷ Methoden: ◇ `next()` - Liefert ein noch nicht geliefertes Element der `Collection` - Reihenfolge von Interface abhängig (`Collection` oder `List`) ◇ `hasNext()` - `true`, falls mindestens ein Element noch nicht durch diesen `Iterator` zurückgeliefert wurde

Interface `Map`	▷ z.B.: `Map<String,Integer> map = new HashMap<String,Integer>();`
	◇ Erster Typparameter: `Key` (hier: `String`)
	◇ Typparameter: `Value` (hier: `Integer`)
	▷ Eine `Map` realisiert eine Abbildung von den `Keys` in die `Values`
	◇ `Keys` müssen alle unterschiedlich sein
	▷ Methoden:
	◇ `put(key, value) // Fügt Paar in Map ein`
	◇ `get(key) // 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 du
	◇ (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 rückwarts 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`

3 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



4 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`

5 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

6 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 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

7 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,..

8 Files

System Properties (java.lang.System)	▷ Attribute der Umgebung, in denen das Java Programm abläuft ▷ Methoden: ◇ `getProperty` - Erhält `String` und gibt `String` zurück ◇ z.B.: `String homeDir = System.getProperty("user.home");` ◇ Mögliche Strings: - `"user.home" // Home directory` - `"user.dir" // Working directory` - `"user.name" // Account name` - `"file.separator" // Zeichen zur Dateitrennung` - `"line.separator" // Zeichen zur Zeilentrennung` ▷ `System.out`: ◇ Klassenattribut `out` von System ist von Klasse `PrintStream` ◇ `PrintStream` hat also auch Methoden wie `println` ▷ `System.err`: ◇ Auch `err` ist von Klasse `PrintStream` ◇ Hierhin werden die Fehlerausgaben geschrieben ◇ z.B. sinnvoll um Fehler in seperate Log-Datei umzuleiten ▷ `System.in`: ◇ Auch `in` ist von Klasse `PrintStream` ◇ Liest Tastatureingaben ▷ Diese drei Attribute können auch auf andere Streams gesetzt werden ◇ z.B.: andere `FileInputStreams/FileOutputStreams` ◇ `System.setIn(in); System.setOut(out); System.setErr(err);`
Klasse Path / Paths	▷ Beide in `java.nio.file` ▷ Objekt der Klasse `Path` verwaltet einen Pfadnamen ◇ Dort muss nicht unbedingt etwas existieren ▷ `Paths` wird nur dazu genutzt um Objekt von `Path` zu erzeugen ◇ z.B.: `Path path = Paths.get(homeDir, "fop.txt");`

Klasse Files

▷ Aus Package `java.nio.file`
▷ Nützliche Sammlung von Klassenmethoden rund um Dateien
▷ Methoden:
◇ `lines // Files.lines(path);`
- Öffnet Datei an übergebenem Pfad
- Liefert einen Stream von Strings, ein String pro Zeile
- Zeilenende durch `"file.separator"` gekennzeichnet
- `IOException`, falls Problem beim Öffnen der Datei (`java.io`)

◇ `exists // Files.exists(path);`
- `true`, wenn es dort Datei/Verzeichnis gibt

◇ `isReadable(path)`
- Fragt lesende Zugriffsrechte ab

◇ `isWritable(path)`
- Fragt schreibende Zugriffsrechte ab

◇ `isRegularFile(path)`
- `true`, falls es eine reguläre Datei ist (kein Verzeichnis)

◇ `isDirectory(path)`
- `true`, falls es ein Verzeichnis ist

◇ `size(path) // long size = Files.size(path);`
- Fragt die Größe der Datei ab
- `long`, da die Dateigröe oft nicht in `int` passt

◇ `createFile(path)`
- Richtet Datei an der übergebenen Stelle ein

◇ `copy(path1, path2)`
- Kopieren von Pfad 1 nach Pfad 2

◇ `move(path1, path2)`
- Umbenennen einer Datei, oft auch Bewegen genannt

◇ `delete(path)`
- Entfernen einer Datei
- `NoSuchElementException`, falls nicht vorhanden

◇ `deleteIfExists(path)`
- Falls das Objekt nicht existiert, passiert garnichts

Beispiel: Einlesen einer Datei in einen String

```
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 }
```

▷ `try-with-resources` wird für Interface `AutoCloseable` verwendet

Bytedaten

▷ Direkt, ohne Bezug zu Streams
▷ Klassen und Interfaces finden sich in `java.io`
▷ Byteweise Verarbeitung sinnvoll für Audio oder Bilddateien, nicht für Text
▷ Wird aber meist durch Bibliotheken oder Ähnliches gehandhabt

Bytedaten lesen

▷ Verwendung eines `InputStream`-Objekts
▷ `InputStream` abstrakt, deswegen nur Subtypen z.B.: `FileInputStream`
◇ `FileInputStream` nutzt den Namen der Datei als String im Konstruktor
▷ Methoden:
◇ `read()`
- Liest nächstes Byte in ein `int`
- Überprüfung, ob -1 um zu prüfen, ob Dateiende erreicht ist

▷ Beispiel:

```
1 FileInputStream in = new FileInputStream (fileName);
2 int n = in.read();
3 if (n == 1) return;
```

Bytedaten schreiben	▷ Verwendung eines `OutputStream`-Objekts
	▷ `OutputStream` abstrakt, deswegen nur Subtypen z.B.: `FileOutputStream`
	◇ `FileOutputStream` 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 `boolean` als zweiten Paramter:
	- Falls `false`: Verhält sich wie normaler Konstruktor
	- Falls `true`: Inhalt geht nicht verloren, wird hinten angehangen
	▷ Methoden:
	◇ `write()`
	◇ Hat `int` als formalen Parametertyp
	◇ Schreibt nur unterestes Byte dieses `int`
	▷ Beispiel:
	`1 FileOutputStream out = new FileOutputStream(fileName);`
	`2 int i = 5;`
	`3 out.write(i);`

Relevante Subtypen von Input-/OutputStream	▷ Geschwindigkeit beim Lesen/Schreiben ist relevant ▷ `BufferedInputStream`: ◇ liest mehrere Bytes auf einmal ein ◇ Konstruktor: `BufferedInputStream(InputStream in)` ◇ Verwendet im Konstruktor z.B. einen `FileInputStream` ▷ `BufferedOutputStream`: ◇ Schreibt zuerst in internen Puffer ◇ Falls dieser voll ist, wird in die Datei geschrieben ◇ Konstruktor: `BufferedOutputStream(OutputStream out)` ◇ Schreibt die Daten auf den OutputStream im Parameter ▷ `PrintStream`: ◇ Ersatz für OutputStream im Package `java.io` ◇ Konstruktor: `PrintStream(OutputStream out)` ◇ Dient als Konvertierer von primitiven Datentypen und String in die byteweise Darstellung ◇ Das eigentliche Schreiben übernimmt der übergebene `OutputStream` ◇ Methode `print` - z.B.: `out1.print(pi = "); out1.print(3.14);` - Byteweise Ausgabe von übergebenen Werten ◇ `System.out.print()`: `out` ist von Klasse `PrintStream` ◇ Methode `println` - Ausgabe von Werten mit Zeilenumbruch
Mehr Subtypen von Input-/OutputStream	▷ `java.util.zip.ZipInputStream` ◇ Zum Einlesen von komprimierten Zip-Dateien ▷ `java.util.jar.JarInputStream` ◇ Zum Einlesen von Jar-Dateien ◇ Jar-Dateien enthalten kompilierte Java-Dateien, mit zip komprimiert ▷ `javax.sound.sampled.AudioInputStream` ◇ für Audio-Dateien ▷ `java.io.PipedInputStream / java.io.PipedOutputStream` ◇ Zwei aneinander gekoppelte Lese/Schreib-Klassen
Textdaten direkt	▷ Bequemere Zugriffsmöglichkeiten für Textdaten vorhanden ▷ `Reader` und `Writer` aus Package `java.io` ▷ Textdatei besteht aus einzelnen Zeichen aka `char` ◇ Jedes `char` ist zwei Byte groß

Textdaten lesen

▷ Komplett analog zu `InputStream` und `FileInputStream`
▷ `Reader` abstrakt, deswegen nur Subtypen z.B. `FileReader`
  ◇ `FileReader` nutzt den Namen der Datei als String im Konstruktor
▷ Methoden:
  ◇ `read`
  - Liest `char`-Werte ein
  - Verschiedene Implementationen z.B.: kein Parameter → einzelner `char`
  - Mit `char-Array`: Liest soviele ein, bis `Array` voll ist
▷ Beispiel:

```
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
```

▷ `BufferedReader`
  ◇ Konstruktor: `BufferedReader(Reader in)`
  ◇ Methode `readLine();`
  - Liest alles vom letzten gelesenen Zeichen bis zum Zeilenende
  - Also meist eine ganze Zeile
▷ Verknüpfung mit byteweisem Einlesen:
  ◇ evtl. sinnvoll, falls offener `InputStream` auf Text-Datenquelle
  ◇ Die Brücke bildet hier der Subtyp `InputStreamReader`

```
1 InputStream in = ...;
2 Reader reader = new InputStreamReader(in);
```

Textdaten schreiben

▷ `Writer` abstrakt, deswegen nur Subtypen z.B. `FileWriter`
  ◇ `FileWriter` benutzt den Namen der Datei als String im Konstruktor
▷ Methoden:
  ◇ `write`
  - Schreibt einzelnen `char` oder ganzen `String`
▷ Beispiel:

```
1 FileWriter writer1 = new FileWriter(fileName);
2 writer1.write('H');
3 writer1.write("ello World");
```

▷ Verknüpfung mit byteweisem Schreiben:
  ◇ Die Brücke bildet hier der Subtyp `OutputStreamWriter`

```
1 OutputStream out = ...;
2 Writer writer = new OutputStreamWriter(out);
```

9 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> // z.B.: 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

10 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`

11 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`

12 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

13 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)

14 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;`

15 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

16 Optional (java.lang.Optional;)

Informationen	▷ Objekt der Klasse `Optional` kapselt ein Objekt seines Typparameters ein ▷ Bietet bequemem Umgang mit der Möglichkeit, dass eine Referenz `null` ist
Methoden	◇ `ofNullable` - Bekommt ein Objekt oder `null` übergeben und kapselt dieses ein - Gibt ein Objekt der Klasse `Optional` zurück ◇ `get` - Liefert das eingekapselte Objekt zurück - Falls `null`: `NoSuchElementException` ◇ `orElseGet` - Zurücklieferung eines anderen Wertes vom Typparameter, falls `null` - Formaler Parameter: `java.util.function.Supplier;` ◇ `ifPresent` - 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` - Abbildung basierend auf Paramter - z.B.: `Optional<Number> opt2 = opt1.map(x -> x * x);` - z.B.: Hier `opt2` auch `null`, da `opt1 == null` ◇ `filter` - 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	▷ `Optional<Number> opt1 = Optional.ofNullable(null);`
	▷ `Number n1 = opt1.get(); // NoSuchElementException`
	▷ `Number n2 = opt1.orElseGet(() -> 0); // Falls null -> 0`

17 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

18 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

19 Random (java.util.Random;)

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

20 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

21 Streams (java.util.stream.Stream;)

Information	▷ Generisches Interface `Stream` ▷ Einheitliche Schnittstelle für Listen, Arrays, Dateien ▷ Relevante Kapitel: `Optional`
Methodenzusammenfassung	▷ filter, map, max, of ▷ `filter` ◇ Liefert Stream vom selben generischen Typ zurück ◇ Formaler Parameter: `java.util.function.Predicate;` ▷ `map` ◇ Liefert Stream von evtl. anderem Typparameter zurück ◇ Dieser Typ ist abhängig vom aktualen Parameter ◇ Formaler Parameter: `java.util.function.Function;` ▷ `max` ◇ Liefert nur einzelnes Element zurück abhängig vom `Comparator` ▷ `of` ◇ Dient der direkten Erzeugung von Streams ◇ Beliebige Anzahl an Parametern des Typarameters ◇ Rückgabe eines Streams mit diesen Elementen ◇ z.B.: `Stream<Number>.of(new Integer(2), new Integer(3));` ▷ `reduce` ◇ Erstellt aus allen Elementen des Streams ein einzelnes Ergebnis ◇ Durch sukzessiven Aufruf der Funktion im aktualen Parameter ◇ z.B.: `String fileContent = stream.reduce(String::concat);`

Stream aus Liste	▷ `List<Number> list = new LinkedList<Number>(); // Erstellt Liste`
	▷ `Stream<Number> stream1 = list.stream();`
	◇ Liefert Stream vom selben generischen Typ
	◇ Methode der Klasse `List`
	▷ `... stream1.filter(myPred); // Anwenden eines Filter`
	▷ `... stream1.map(myFct); // Anwenden einer Abbildung`
	▷ `Optional<Number> opt = stream.max(new MyComp());`
	◇ 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:
	▷ `... = list.stream().filter(myPred).map(myFct).max(new MyComp());`

Stream aus Array	▷ `Number[] a = new Number[100]; // Erstellt Array`
	▷ `Stream<Number> stream1 = Arrays.stream(a); // Erzeugt Stream`
	◇ Aufruf der `Arrays`-Klassenmethoden `stream(Array a)`

Iterator	▷ `Iterator iter = stream.iterator(); // Erzeugt Iterator Objekt`
	▷ `iter.hasNext() // Verwendung als Abbruchbedingung`
	▷ `iter.next() // Zum Fortschreiten im Iterator`

Liste aus Stream	▷ `List<String> list = stream.collect(Collectors.toList());` ◇ `Collectors` besitzt viele Klassenmethoden zur Verarbeitung von Streams ◇ `toList()` liefert das generische Interface `Collector`
Array aus Stream	▷ `Number[] a = stream.toArray(Number[]::new);` ▷ Art der Erzeugung abhängig vom Parameter ▷ Parameter: Siehe Methodennamen als Lambda-Ausdrücke
Int-/Long-/ DoubleStreams	▷ Methoden sind genau diesselben wie bei normalen Streams ▷ z.B.: `IntStream stream1 = IntStream.of(1,2,3);` ▷ Nutzen der Klasse `Random` für unendlichen Stream mit Zufallszahlen ◇ `IntStream stream1 = new Random().ints();`

22 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`

23 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`

24 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