FOP Reference Sheet

Jonas Milkovits

Last Edited: 28. April 2020

Inhaltsverzeichnis

1 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

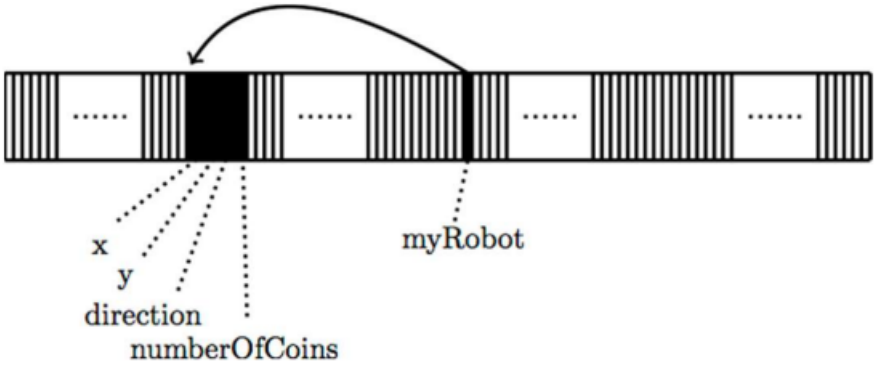
Interface `Iterator`	▷ `Collection` und `List` erben von Interface `Iterable` ▷ Jede Klasse, die `Collection` implementiert hat eine eigene `Iterator`-Klasse ▷ Diese eigene `Iterator`-Klasse implementiert das Interface `Iterator` ▷ `Collection<Number> c1 = new ArrayList<Number>();` ▷ `Iterator<Number> 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 durch alle Elemente: (**LOGIK**)
	◇ (Die eigentliche Implementation in Java sieht anders aus)
	◇ `for (ListItem<T> p = head; p != null; p = p.next) {...}`
	◇ Setzen von `p` zu `p.next` bis `p == null`
	▷ Einfügen Element am Anfang: (**LOGIK**)
	◇ Erstellen eines neuen Listitems und Kopieren der Werte
	◇ Achtung: Erst `head` als `next` abspeichern
	◇ Danach neues Listitem als `head` setzen
	◇ (sonst geht die komplette Liste verloren)
	▷ Einfügen Element an Stelle n: (**LOGIK**)
	◇ Fortschreiten des Durchlaufs bis zu n-1
	◇ `ListItem<T> tmp = new ListItem<T>();`
	◇ `tmp.key = key; // Setzen des Keys`
	◇ `tmp.next = p.next; // Knüpfen des neuen Elements an n+1.Element`
	◇ `p.next = tmp; // Knüpfen des n-1.Elements an neues Element`
	▷ Entfernen Element: (**LOGIK**)
	◇ Überspringen des zu löschenden Elements
	◇ `head`: `head = head.next;`
	◇ Sonst: `p.next = p.next.next;`
	- Laufpointer muss in diesem Fall eine Stelle davor stehenbleiben
	▷ Allgemein:
	◇ Auf korrektes Zwischenspeichern achten!
	▷ Doppelte Verkettung:
	◇ Ermöglicht 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`

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 und übersichtlich ◇ Können zusätzlich vom Compiler an- und abgeschaltet werden ◇ z.B.: Verwendung für Tests für Methoden und späteres Abschalten ▷ Solche Tests werden White-Box-Tests genannt
Exceptions aus Lambda-Ausdrücken	▷ Gute Verwendung, falls die funktionale Methode eines Interface Fehler wirft ▷ `MyFuncIntf fct = (m,n) ->` `{if (n == 0) throw new Exception(); return m/n;};` ▷ Funktionale Methode könnte führt Berechnung aus ▷ Falls `n == 0` wird jedoch eine Exception geworfen

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 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);
```

8 Functional Interfaces und Lambda-Ausdrücke

Functional Interface

- ▷ Interface, bei dem genau eine Methode weder `default` oder `static` ist
  - ◇ Diese Methode heißt funktionale Methode dieses `Functional Interface`
- ▷ Jedes `Functional Interface` hat genau eine funktionale Methode
- ▷ `Functional Interface IntToDoubleFunction`
  - ◇ Aus Package `java.util.function`
  - ◇ Hat die funktionale Methode `double applyAsDouble(int n);`

Lambda-Ausdrücke Einführung

- ▷ Sind Literale von Funktionstypen
- ▷ Abgekürzte Schreibweise für den Aufruf der Hauptmethode eines Func. Interface
- ▷ Verwendung am Beispiel `IntToDoubleFunction` **ohne** Lambda:

```
IntToDoubleFunction fct1 = new X(2);
double y = fct1.applyAsDouble(10);
```

  - `X` ist hier eine Klasse, die das Interface und die Methode implementiert
- ▷ Verwendung am Beispiel `IntToDoubleFunction` **mit** Lambda:

```
IntToDoubleFunction fct2 = x -> x * 10;
double z = fct2.applyAsDouble(10);
```

  - Richtet nicht sichtbare Klasse ein, die Interface implementiert
  - Lambda-Ausdruck wird für die funktionale Methode verwendet
  - Speichern der Referenz eines Objektes dieser Klasse in `fct2`

Closure

- ▷ Falls der Parameter (oben 10) zur Laufzeit nicht feststeht (z.B. y):
  - ◇ Unsichtbare Klasse erhält Attribut, das durch Konstruktor initialisiert wird
  - ◇ Verwendung dieses Attributs innerhalb der Klasse
- ▷ Info aus Entstehungskontext des Lambda-Ausdrucks wird mitgespeichert
  - ◇ Aktualer Wert aber nicht unbedingt kopiert, sondern referenziert
  - ◇ Vorsicht bei Änderungen

Lambda-Ausdrücke Aufbau

- ▷ `n -> n 70 % 2 == 1`
  - ◇ Kurzform
- ▷ `(int n) ->{return n 70 % 2 == 1;}`
  - ◇ richtige lange Form, Typangabe des Parameters optional
- ▷ `(int n, double x) -> {System.out.print(x); System.out.print(n);}`
  - ◇ Langform ermöglicht mehrere Anweisungen

Beispiel Prädikate

- ▷ Prädikat: boolsche Funktione, die entweder `true` oder `false` zurückliefert
- ▷ Gut für Methoden, die z.B. variablen Filter implementieren
- ▷ `java.util.function.IntPredicate`:
  - ◇ Funktionale Methode `boolean test(int x);`
  - ◇ Beispiel:
    - `IntPredicate pred1 = n -> n 70 % 2 == 1`
    - Gibt `true` zurück, falls `n` ungerade ist
- ▷ `java.util.function.IntPredicate` hat noch `default`-Methoden:
  - ◇ Zugriff auf diese über `pred1`:
    - `IntPredicate pred4 = pred1.negate();`
    - Die Klasse des Objekts `pred1` implementiert ja das Interface
  - ◇ Ergänzung des `Functional Interface` durch diese `default`-Methoden ▷ Natürlich auc
  - ◇ `IntPredicate[] predicates = new IntPredicate[6];`
  - ◇ `predicates[0] = n -> n > 0;`
- ▷ Auch Erstellung einer Liste möglich
  - ◇ `List<IntPredicate> pred = new ArrayList<IntPredicate>();`
  - ◇ `pred.add(n -> n > 0);`

Methodennamen als Lambda

- ▷ Für Lambda-Ausdrücke, die nur aus einzelnem Methodenaufruf bestehen
  - ◇ Fachbegriff `method reference`
- ▷ Normalerweise: `... = x -> {System.out.print(x);}`
- ▷ Hier: `... = System.out::print;`
  - ◇ Verbinden des Methodennamens mit Referenz auf Objekt durch Doppelpunkt
  - ◇ Funktioniert auch analog mit Klassenmethoden

9 Graphical User Interface

Window Manager	▷ Systemprozess, der permanent im Hintergrund als `Service` läuft ▷ Stellt generelle, anwendungsunspezifische Funktionalitäten zur Verfügung ◇ Öffnen, Schließen, Ikonifizieren, Größe ändern ◇ Rahmen um Fenster, Bildschirmhintergrund

Klasse Frame	▷ Abgeleitet von `java.awt.Window;` (awt = abstract window toolkit)
	▷ Im Gegensatz zu `Window` aber mit Rahmen (vom `Window Manager` verwaltet)
	▷ Beispielkonstruktor: `Frame frame = new Frame(string); // Fenstertitel`
	▷ Vorhergehensweise:
	◇ Erstellung einer Klasse, die `Frame` erweitert
	◇ Hinzufügen von Funktionalitäten
	▷ Methoden:
	◇ `setVisible(boolean b)`
	- `Frame` ist entweder sichtbar oder unsichtbar
	- Standardmäßig unsichtbar
	- Erst Fenster aufbauen, dann sichtbar machen
	◇ `setBackground(Color bgColor)`
	- Setzt die Hintergrundfarbe des Fensters
	◇ `dispose()`
	- Alle Ressourcen des Fensters und der Bestandteile werden freigegeben
	◇ `setExtendedState(int state)`
	- Setzt den Status des Fensters
	- `ICONIFIED`: Ikonifiziert das Fenster
	- `NORMAL`: Deikonifiziert das Fenster
	- `MAXIMIZED_HORIZ`: Ausbreitung auf gesamte Horizontale
	◇ `add(Component comp)`
	- Fügt den übergebenen Komponenten zum `Frame` hinzu

Komponenten	▷ Eigene Klasse für jede Komponente ▷ Alle Klassen oder Interfaces aus `java.awt`, falls nicht anders gesagt ▷ Werden mithilfe von `add(Component comp)` zum Fenster hinzugefügt
Button	▷ Konstruktor: `Button(String label) // Text auf dem Button` ▷ Methoden: ◇ `setFont(Font f)` - Zum Setzen der Schriftart - Konstruktor `Font: Font(String name, int style, int size)` ◇ `addActionListener(ActionListener l)` - Fügt den übergebenen `ActionListener` hinzu - Bei jedem Klick wird `actionPerformed` des `Listeners` aufgerufen - Auch mehrere möglich - Automatische Einrichtung des `Event Dispatch Thread` ◇ `setLabel(String label)` - Setzt den Titel des `Button`

Interface ActionListener	▷ Zugehörig zu `Button` ▷ Aus Package `java.awt.event` ▷ Funktionales Interface ▷ Funktionale Methode `actionPerformed (ActionEvent event)` ▷ Vorhergehensweise: ◇ Erstellen einer eigenen Klasse, die `ActionListener` implementiert ◇ Erstellen relevanter Attribute und Konstruktor für gegebenen Fall ◇ Implemetieren der Methode `actionPerformed (ActionEvent event)` ◇ Erstellen eines Objekts unserer Klasse - `ActionListener listener = new MyListener(frame);` ◇ Hinzufügen des `Listener` zum `Button` - `button.addActionListener(listener);` ▷ Alternativ: ◇ Erstellung des `Listener` in der Subklasse des `Frame` - Keine `Frame`-Übergabe notwendig - z.B.: als `private`-Klasse (Stichwort: `nested classes`) ▷ Da `Functional Interface`: Lambda-Ausdruck ◇ `button.addActionListener( (e) -> {System.out.print("Hello");})`
Klasse ActionEvent	▷ Übergebener Parameter bei `actionPerformed` ▷ Methoden: ◇ `getWhen()` - Gibt die Uhrzeit des Geschehnisses als `long` zurück - Nützlich: `java.sql.Timestamp` - `Timestamp stamp = new Timestamp (event.getWhen());` - Methoden: `stamp.getHour(); stamp.getMinute();`
Übersicht Listener und Events	▷ `Listener`-Interface ↔ `Event`-Klasse ▷ `KeyListener` ↔ `KeyEvent` ▷ `MouseListener` ↔ `MouseEvent` ▷ `MouseMotionListener` ↔ `MouseEvent` ▷ `MouseWheelListener` ↔ `MouseWheelEvent` ▷ `WindowFocusListener` ↔ `WindowEvent` ▷ `WindowListener` ↔ `WindowEvent` ▷ `WindowStateListener` ↔ `WindowEvent` ▷ Hinzufügen: ◇ `addKeyListener(...)` ◇ `addMouseListener(...)` ◇ `addWindowListener(...)`
Adapter	▷ Verwendung von Adaptern, wenn passendes Interface nicht `functional` ist ◇ z.B. Interface `KeyListener, MouseListener,..` ◇ Diese Interfaces besitzen mehrere Methoden ▷ Adapter sind Klassen und bestehen zu jedem `Listener`-Interface ◇ z.B.: `KeyAdapter, MouseAdapter` ◇ Diese Adapter implementieren das dazugehörige Interface ◇ Die Methoden werden jedoch leer gelassen ▷ Vorteil vom Adapter: ◇ Nicht alle Methoden müssen implementiert werden ◇ Nur die genutzten Methoden (z.B.: `keyPressed()`) werden implementiert ▷ Verwendung: ◇ Erweitern der eigenen `Listener`-Klasse mit Adapter ◇ z.B.: `public class MyKeyListener extends KeyAdapter {...}`

Interface KeyListener

▷ Abhorchen der Tastatur
▷ Erstellen eigener Klasse, die die Klasse `KeyAdapter` (siehe Adapter) erweitert
▷ Methoden:
◇ `public void keyPressed (KeyEvent event)`
- Wird beim Herunterdrücken einer Taste ausgeführt
◇ `public void keyReleased (KeyEvent event)`
- Wird beim Loslassen einer Taste ausgeführt
◇ `public void keyTyped (KeyEvent event)`
- Wird beim Antippen einer Taste ausgeführt

Klasse KeyEvent

▷ Übergebener Parameter bei z.B.: `keyPressed`
▷ Methoden:
◇ `getKeyCode()`
- Liefert die Kodierung der gedrückten Taste zurück
▷ Klassenkonstanten für jede Taste:
◇ z.B.: `KeyEvent.VK_A // Buchstabe A`
◇ z.B.: `KeyEvent.VK_COLON // Doppelpunkt`
◇ z.B.: `KeyEvent.VK_BACKSPACE // Backspace Taste`
▷ Beispiel Verwendung:

```
1 public class MyKeyListener extends KeyAdapter {
2   public void keyPressed (KeyEvent event) {
3     switch (event.getKeyCode()) {
4       case KeyEvent.VK_A: ... break;
5       case KeyEvent.VK_COLON: ... break;
6       case KeyEvent.VK_Backspace: ... break;
7     }
8   }
9 }
```

Interface MouseListener

▷ Abhorchen der Maus
▷ Erstellen eigener Klasse, die die Klasse `MouseAdapter` erweitert
◇ `MouseAdapter` implementiert alle drei `Mouse`-Interfaces
◇ `MouseListener, MouseMotionListener, MouseWheelListener`
▷ Methoden:
◇ `public void mouseClicked (MouseEvent event)`
- Wird beim kurzen Klicken der Maustaste ausgeführt
◇ `public void mousePressed (MouseEvent event)`
- Wird beim Herunterdrücken der Maustaste ausgeführt
◇ `public void mouseReleased (MouseEvent event)`
- Wird beim Loslassen der Maustaste ausgeführt
◇ `public void mouseEntered (MouseEvent event)`
- Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich betritt
◇ `public void mouseExited (MouseEvent event)`
- Wird ausgeführt, sobald der Mauszeiger den abgehorchten Bereich verlässt

Interface MouseMotionListener

▷ Abhorchen der Mausbewegung
▷ Methoden sind auch in Klasse `MouseAdapter` enthalten
▷ Methoden:
◇ `public void mouseDragged (MouseEvent event)`
◇ `public void mouseMoved (MouseEvent event)`

Interface MouseWheelListener

▷ Abhorchen der Mausradbewegung
▷ Methoden sind auch in Klasse `MouseAdapter` enthalten
▷ Methoden:
◇ `public void mouseWheelMoved (MouseWheelEvent event)`

Klasse MouseEvent	▷ Übergebener Parameter bei z.B.: `mouseClicked` ▷ Methoden: ◇ `getButton()` - Liefert die gedrückte Taste zurück ◇ `getX()` - Liefert x-Koordinate abhängig vom Ursprung des Bereichs ◇ `getY()` - Liefert y-Koordinate abhängig vom Ursprung des Bereichs ▷ Klassenkonstanten für Maustasten: ◇ `MouseEvent.BUTTON1` ◇ `MouseEvent.BUTTON2`
Klasse MouseWheelEvent	▷ Übergebener Parameter bei z.B.: `mouseWheelMoved` ▷ Methoden: ◇ `getWheelRotation()` - Liefert die Anzahl der gedrehten Ticks"
Interface WindowListener	▷ Abhorchen von Fensteraktionen ▷ Erstellen eigener Klasse, die die Klasse `WindowAdapter` erweitert ◇ `WindowAdapter` implementiert alle drei `Window`-Interfaces ◇ `WindowListener, WindowStateListener, WindowFocusListener` ▷ Methoden: ◇ `public void windowOpened (WindowEvent event)` ◇ `public void windowClosing (WindowEvent event)` ◇ `public void windowClosed (WindowEvent event)` ◇ `public void windowClosed (WindowEvent event)` ◇ `public void windowDeactivated (WindowEvent event)` ◇ `public void windowIconified (WindowEvent event)` ◇ `public void windowDeiconified (WindowEvent event)`
Interface WindowStateListener	▷ Abhorchen des Status des Fensters ▷ Methoden sind auch in `WindowAdapter` vorhanden ▷ Methoden: ◇ `public void windowStateChanged ( WindowEvent event )`
Interface WindowFocusListener	▷ Abhorchen des Fokus im Bezug auf das Fenster ▷ Methoden sind auch in `WindowAdapter` vorhanden ▷ Methoden: ◇ `public void windowGainedFocus ( WindowEvent event )` ◇ `public void windowLostFocus ( WindowEvent event )`

Klasse Canvas	▷ abgegrenzte Zeichenfläche in einem Fenster ▷ Vorhergehensweise: ◇ Erstellung eigener Subtyp-Klasse von `Canvas` ◇ Implementieren der Methode `public void paint (Graphics graphics)` ◇ Füllen der Methode mit eigener Zeichenlogik ◇ Verwendung von `java.awt.Graphics;` ◇ Hinzufügen zum `Frame` mithilfe von `add` ▷ Beleuchtung nützlicher Aspekte von Graphics: ▷ `FontMetrics` ◇ Informationen über festgelegte Schriftart und Schriftgröße ◇ Abfrage: - `FontMetrics fontM = graphics.getFontMetrics();` ◇ Abfrage der maximalen Stringhöhe: - `int maxHeight = fontM.getMaxAscent() + fontM.getMaxDescent();` - Methoden geben maximalen Abstand von der Basislinie des Textes an ◇ Abfrage der Stringbreite von gegebenem String: - `int widthStr = fontMetrics.stringWidth(string);` ▷ Abfrage des Zeichenfensters als Rechteck: - `Rectangle area = graphics.getClipBounds();` - `x` und `y` geben den Ursprung an - `width` und `height` die Breite und Höhe ▷ Einige Methoden von `Graphics` ◇ `setColor(Color color)` ◇ `fillOval(...)` ◇ `drawOval(...)` ◇ `drawString(...)`
Klasse Checkbox	▷ Kleiner Button (Pin) mit etwas Text ▷ Zwei Zustände: An oder Aus ▷ Konstruktor: ◇ `Checkbox(String label) // Titel der Checkbox` ◇ `Checkbox` standardmäßig aus ▷ Benötigt ein Objekt vom Typ `ItemListener` (siehe unten) ◇ `ItemListener item = new MyItemListener(checkbox,...);` ▷ Methoden: ◇ `isSelected()` - `true`, wenn die `Checkbox` an ist ◇ `setLabel(string);` - Setzt den Titel der `Checkbox`
Interface ItemListener	▷ Verwendung bei `Checkbox` und `Choice` ▷ Funktionales Interface ▷ Funktionale Methode `itemStateChanged (ItemEvent event)` ▷ Vorhergehensweise analog zu `ActionListener` ◇ Erstellung neuer Klasse, die `ItemListener` implementiert

Klasse Choice	▷ Repräsentiert ein Auswahlmenü ▷ Verwendet auch das Interface `ItemListener` ▷ Konstruktor: ◇ `Choice choice = new Choice();` ▷ Methoden: ◇ `add(string)` - Hinzufügen neuer Auswahlen - Startet bei Index 0 ◇ `select(int)` - Legt eine Auswahl als Standard fest - Übergabe des Index als `int` ◇ `getSelectedItem()` - Liefert den ausgewählten String zurück ◇ `getSelectedIndex()` - Liefert Index der aktiven Auswahl
Klasse Label	▷ Nicht durch User interagierbares Rechteck mit Text ▷ Konstruktor: ◇ `Label(String text) // Labeltext` ▷ Wartet auf Events bei anderen Entitäten ▷ Methoden: ◇ `setAlignment(int alignment)` - Auswahl der Zentierung des Textes - Paramter: `Label.CENTER, Label.RIGHT, Label.LEFT` ◇ `setBackground(Color c)` - Setzen der Hintergrundfarbe ◇ `setText(String text)` - Setzt den Text des `Label` - z.B.: Aufruf beim Drücken eines `Button`
Klasse List	▷ Auswahlmenü ▷ Aus `java.awt`, **nicht** `java.util` ▷ Konstruktor: ◇ `List(int rows, boolean multipleMode)` ◇ `rows` gibt die maximale Anzahl der zugleich angezeigten Menüpunkte an ◇ Anzahl der Möglichkeiten größer als `rows` → Scrollbar ◇ `multipleMode`: Auswahl mehrerer Menüpunkte ermöglichen ▷ Methoden: ◇ `add(String item)` - Hinzufügen neuer Menüpunkte ◇ `getSelectedIndexes()` - Liefert die Indizes der ausgewählten Punkte ◇ `setMultipleMode(boolean b)` - De-/Aktivieren der Mehrfachauswahl
Klasse Scrollbar	▷ Werden meist automatisch hinzugefügt (`List`) ▷ z.B.: Erstellung eines eigenen Schiebereglers ▷ Benötigt ein Objekt vom Typ `AdjustmentListener` (siehe unten) ◇ z.B.: `AdjustmentListener adjust = new MyAdjustListener(frame);` ▷ Konstruktor: ◇ `Scrollbar(int orientation, int value, int visible, int minimum, int maximum)` ◇ `orientation`: `Scrollbar.VERTICAL, Scrollbar.HORIZONTAL` ◇ `value`: Startwert der Scrollbar ◇ `visible`: Größe des scrollbaren Balkens ◇ `minimum`: Minimal einstellbarer Wert ◇ `maximum`: Maximal einstellbarer Wert

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

Hierarchie graphischer Komponenten	▷ Vom `java.awt.Component` direkt abgeleitet: ◇ `Button` ◇ `Canvas` ◇ `Checkbox` ◇ `Choice` ◇ `Label` ◇ `List` ◇ `Scrollbar` ◇ `TextComponent // Supertyp von TextArea und TextField` ◇ `Container` ▷ Von `Container` direkt abgeleitet: ◇ `Window` ▷ Von `Window` direkt abgeleitet: ◇ `Frame`
Klasse Component	▷ Die meisten Methoden sind hier definiert, aber nicht implementiert ◇ z.B.: `setVisible(boolean b), setFont(Font f),..` ◇ Die Methoden werden in den Komponentenklassen dann implementiert
Klasse Container	▷ Fasst mehrere Komponenten zu einer zusammen ▷ Hinzufügen von `Buttons,..., Windows, Frames, Containern` möglich ▷ **Wichtig:** Hinzufügen von `Container` möglich ◇ Ähnliche Struktur wie ein Ordnerverzeichnis ◇ z.B.: `Frame` in einem `Frame` ▷ Methoden: ◇ `paint (Graphics graphics)` - In `Component` definiert, hier überschrieben - Ruft `paint` für jeden enthaltenen Komponenten auf ◇ `add (Component comp)` - Hinzufügen einer Komponente zum `Container` ◇ `add (Component comp, Object constraints)` - Steuerung der Position mithilfe des zweiten Parameters - Weiteres bei `LayoutManager` ◇ `setLayout (LayoutManager manager)` - Setzen des `LayoutManager` - Dieser steuert die Platzierung der Komponenten - Jeder `Container` hat zu jedem Zeitpunkt einen `LayoutManager` ◇ `validate()` - Aktualisierung nach z.B.: Größenänderung

Klasse LayoutManager	▷ Wird bei Erstellung eines `Containers` oder Subtyps automatisch eingerichtet ◇ Standardklasse für für `Window` und `Frame` ist `BorderLayout` ▷ `BorderLayout` ▷ Einteilung des Fensters in fünf Bereiche ◇ `NORTH, EAST, SOUTH, WEST, CENTER` ◇ Mögliche Positionen als Klassenkonstaten vordefiniert - z.B.: `BorderLayout.NORTH, BorderLayout.CENTER,..` ◇ Verwendung bei `add (Component comp, Object constraints)` - z.B.: `frame.add (comp1, BorderLayout.NORTH);` - Ohne Wahl der Position (normales `add`): `CENTER` als Standard ▷ `BorderLayout` an sich für das Fenster an sich meist die richtige Wahl ◇ Aber nicht unbedingt für `Container` innerhalb eines Fensters ▷ `BoxLayout` ◇ Anlegen in einer Reihe nacheinander ◇ Wahl ob vertikal oder horizontal im Konstruktor ▷ `GridLayout` ◇ Matrixartiges Anlegen (wie Telefontastatur) ◇ Anzahl Zeilen und Spalten im Konstruktor festgelegt ▷ `BorderLayout, BoxLayout und GridLayout`: ◇ Passen Größe der Komponenten anhand der Gesamtsituation an ◇ Nicht unbedingt passendste Größe für Komponente ▷ `FlowLayout` ◇ Anlegen in einer Zeile nebeneinander - Anfangen einer Zeile, falls die alte voll ist ◇ Wählt automatisch die bestmöglichste Größe für Komponenten - Abfrage über `getPreferredSize()` ▷ `CardLayout` ◇ Zeigt Komponenten nicht alle gleichzeitig, sondern nacheinander ◇ Navigation: `first, last, next, previous` ▷ `validate()` ◇ Notwendig zur Aktualisierung von sichtbaren Fenstern ◇ Wann: - Ändern der Anzahl von Komponenten - Ändern der Größe von Komponenten (auch Schrift)
Java Swing (javax.swing)	▷ `java.awt` als Grundlage ▷ Zweite Bibliothek, die die Funktionalitäten erweitert ▷ Verbindung zu `java.awt`: ◇ `JFrame extends java.awt.Frame` ◇ `JComponent extends java.awt.Container` - Funktionalitäten von `Container` hier in `JComponent` ▷ Von `JComponent` abgeleitet: ◇ `JButton, JCheckbox, JLabel` ◇ `JList<T>, JScrollbar, JTextComponent` ▷ `JButton, JCheckbox` sind aber indirekt abgeleitet: ◇ Zwischenklasse `AbstractButton` bei beiden - `public class JButton extends AbstractButton{}` ◇ Bei `JCheckbox` zusätzlich noch `JToggleButton extends AbstractButton` - `public class JCheckbox extends JToggleButton {}` ◇ **Grund:** - Einführung zusätzlicher, eng verwandter, Komponenten - Deswegen Auslagerung in Supertyp ▷ `JList<T>`: ◇ in `Swing` generisch, Liste von beliebigem Referenztyp ◇ in `java.awt` wird String verwendet

Klasse JComponent

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

```
1 String s = UIManager.getSystemLookAndFeelClassName();
2 UIManager.setLookAndFeel(s);
```

◇ Setzen des Look and Feel auf z.B. Java-Standard
Methode: `UIManager.setLookAndFeel(LookAndFeel lookAndFeel)`

```
1 UIManager.setLookAndFeel(new MetalLookAndFeel());
```

▷ `KeyBindings:`
◇ Funktionalitäten wie `Listener` automatisch umgesetzt
◇ Erläuterung der Funktion anhand eines Beispiels:

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

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

Weitere GUI-Klassen

▷ `JFormattedTextfield`:
- ◇ Erlaubt Formatierungsregeln für den einzugebenden Text
- ◇ z.B. für Datumsangaben ◇ `JFormattedTextField extends JTextField`

▷ `JPasswordField`:
- ◇ Zusätzliche Funktionalitäten für Passwörter

▷ `JRadioButton`:
- ◇ kleiner, anklickbarer Bereich
- ◇ Verwendung im Rahmen eines Objekts von Klasse `ButtonGroup`
- ◇ Nur ein `RadioButton` in `ButtonGroup` kann gleichzeitig angeklickt sein

▷ `JToolbar`:
- ◇ Vereinfachte Möglichkeit für Standardmenüs

▷ `JSlider`:
- ◇ Klasse für Schieberegler
- ◇ Besser als Verwendung einer `Scrollbar` als Schieberegler

▷ `Popup`:
- ◇ Popup-Fenster

▷ `JTable`:
- ◇ Umsetzung einer Tabelle
- ◇ Häufiges Verwendungsbeispiel:

```
1 Object[][] entries = ...;
2 Object[] columnNames = ...;
3 JTable table = new JTable(entries, columnNames);
4 JScrollPane scrollPane = new JScrollPane(table);
5 table.setFillsViewportHeight(true);
6 int[] selectedRows = table.getSelectedRows();
7 int[] selectedColumns = table.getSelectedColumns();
```

- Zeile 1: Erzeugung der Tabellenmatrix
- Zeile 2: Erzeugung der Spaltennamen
- Zeile 3: Konstruktor der Tabelle mit den eben erstellen Arrays
- Zeile 4: `JScrollPane` kapselt Objekt von `Component` ein
- Zeigt nur einen Ausschnitt der übergebenen Komponente
- Fügt außerdem `Scrollbar` ein
- Zeile 5: Vertikale Streckung der Tabelle, um gesamte Höhe auszufüllen
- Zeile 6: Abfrage der momentan ausgewählten Zeile
- Zeile 7: Abfrage der momentan ausgewählten Spalte

10 Generics

Wrapper-Klassen	▷ 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

11 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`

12 Interfaces

Erzeugung	▷ Meist in eigener Datei ▷ `public interface MyInterface {...}` ▷ Alle Methodes und das Interface **müssen** `public` sein
Methoden	▷ Objektmethoden 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 ▷ Klassenmethoden können definiert und implementiert werden - Statischer Typ entscheidet bei Klassenmethoden welche Implementation - Deswegen ist dies hier in `Java` kein Problem ▷ Außerdem möglich: ◇ Klassenkonstanten (sind aber implizit `public` und `final`) ◇ Implementierte "Default"-Objektmethoden

Default-Objektmethoden	▷ Werden durch Schlüsselwort `default` vor Rückgabetyp eingeleitet
	▷ Problem bei Mehrfachvererbung von Interfaces:
	◇ Compiler wirft Fehlermeldung, falls Klasse zwei Implementationen der selben Methode erbt
	▷ Beispiel:
	`1 public interface Int1 {`
	`2 default void m() {...}`
	`3 }`

Verwendung	▷ `implements MyInterface` nach Klassenname ▷ Beliebig viele Interfaces möglich (seperiert durch ,) ▷ Ein Interface kann mehrere andere Interfaces erweitern (`extends`
Unterschiede Interface Abstrakte Klasse	▷ Interfaces können Mehrfachvererbung ▷ Abstrakte Klassen können von Klassen abgeleitet werden ▷ Abstrakte Klassen, können non-`public` Attribute/Methoden haben

13 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

14 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)

Verschachtelte Klassen	▷ Einbettung von Klasse in andere Klasse
	▷ Eingebettete Klasse sind ähnlich einem Attribut einer Klasse
	▷ Zum Beispiel:
	`1 public class X {`
	`2 private class Y {...}`
	`3 }`
	▷ `Y` ist in diesem Fall die innere, `X` die äußere Klasse
	▷ Innere Klasse darf:
	◇ Alle Identifier möglich
	▷ Äußere Klasse darf:
	◇ Nur `public` oder ohne `public`, kein `private` oder `protected`
	▷ Maximal eine Klasse darf `public` sein → Name der Quelldatei
	▷ Innere Klassen sind davon allerdings nicht betroffen
	▷ Objekterzeugung:
	◇ Erstellung von Objekten der inneren Klasse über Objekt der äußeren Klasse
	◇ Automatische Erzeugung eines Verweises auf Erzeugungsobjekt
	◇ `X a = new X(); a.newY();`
	▷ Aufruf:
	◇ `OuterClass.InnerClass x = ...;`
	◇ Äußere Klasse + Innere Klasse durch Punkt getrennt
	▷ `static`:
	◇ `static` auch bei inneren Klassen möglich
	◇ Kann nur auf Klassenmethoden und -attribite zugreifen
	◇ Erzeugung ohne Objekt möglich z.B.: `X.Y a = new X.Y();`

15 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;`

16 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

17 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`

18 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

19 Programme und Prozesse

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

20 Random (java.util.Random;)

Verwendung	▷ Erzeugung eines neuen Objekts ◇ `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

21 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

22 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();`

23 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`

24 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 ▷ Boolesche Bestandteile: Prädikate (`isGreen`) ▷ Andere Bestandteile mit Wert: Beschreibung des Wertes ▷ Subroutinen ohne Rückgabe: Imperativ (`fillOval`)
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`

25 Threads

Interface Runnable	▷ Aus Package `java.lang` ▷ Enthält den Inhalt des parallel laufenden Prozesses ▷ `Functional Interface` mit funktionaler Methode `run` ▷ Funktionsweise: ◇ Erstellung einer Klasse, die das Interface `Runnable` implementiert ◇ Implementierung der funktionalen Methode `run` - `public void run() {...}` ◇ Erzeugung eines Objekts unserer Klasse - z.B.: `Runnable runnable = new MyRunnable(...);` ◇ Erzeugung eines `Thread`-Objekts mithilfe unseres `runnable` - `new Thread(runnable).start();` ◇ Der `Thread` wird dadurch auch gestartet
Klasse Thread	▷ Aus Package `java.lang` ▷ `Thread` organisiert einen parallel laufenden Prozess ▷ Methoden: ◇ `static currentThread` - Keine Parameter - Liefert den `Thread` in dem die Methode aufgerufen wurde ◇ `dumpStack` - Schreiben den CallStacks auf `System.err` ◇ `static getAllStackTraces` - Liefert die CallStacks aller `Threads` als `Map` ◇ `getId` - Jeder `Thread` besitzt eine ID von Typ `long` - Diese ID ist einmalig und bleibt gleich ◇ `getName` - Abfrage des nicht einmaligen Namens ◇ `getPriority; setPriority` - Jeder `Thread` besitzt eine Priorität - Anfangs gesetzt und dauernd beschränkt durch Klassenkonstanten ◇ `static sleep` - Anhalten des `Threads` für übergebene Pause (`long`) ◇ `getState` - Gibt den Status des `Threads` aus
Threads und Streams	▷ Verknüpfung zweier `Threads` mithilfe von `PipedInput(Output)Stream` ▷ z.B.: Ungefähre Vorhergehensweise: ◇ Erzeugung beider Streams: (Beachten von `try-catch`): - `PipedOutputStream out = new PipedOutputStream();` - `PipedInputStream in = new PipedInputStream(out);` ◇ Implementieren einer schreibenden `Runnable`-Klasse - z.B.: Schreiben von zufälligen Zahlen auf `out` ◇ Erstellen des `Threads`: - `Runnable runnable = new MyWriteRunnable(out);` - `new Thread(runnable).start();` ◇ Lesen mithilfe `in` der geschriebenen Daten ⇒ Zwei verbundene Streams, einer schreibt, der andere liest
Interferiende Threads	▷ Reihenfolge der Zugriffe, bei Zugriff auf die selbe Ressource, ungewiss ▷ z.B.: Gleichzeitiges Schreiben auf `StdOut // Standard Out` → `System.out`

Thread terminieren	▷ Beispiel: ◇ Einfügen einer Boolean-Variable in dazugehöriger `Runnable`-Klasse ◇ Ausführung von `run()` solange diese `false` ist ◇ Setzen der Variable auf `true`, wenn terminiert werden soll ▷ Sobald die Methode `run` beendet ist, terminiert der `Thread` ▷ Andere Umsetzung: ◇ Einfügen einer `terminate()`-Methode in die `Runnable`-Klasse ◇ Diese setzt z.B. die oben implementierte Variable auf `true` ◇ Zugriff auf diese Methode über das erzeugte `Runnable`-Objekt
Gründe für Threads	▷ Parallelisierung ◇ Aufteilung der Arbeitslast ◇ Oft jedoch nicht schneller, sondern langsamer ▷ Abspaltung von eigenständigen Programmteilen ◇ Starten und Vergessen
Parallelisierung von Streams	▷ Bereits implementiert, automatische, effiziente Aufteilung ▷ Methode `parallelStream()` ◇ Kann, aber muss nicht, aufteilen ◇ Liefert den selben Stream als Rückgabetyp zurück ◇ bequeme Möglichkeit zur Verarbeitung großer Datenmengen

26 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

27 Anhang: Interne Zahlendarstellung

Ganze Zahlen

Ganzzahlige Datentypen	▷ `byte` 8 Bits ▷ `short` 16 Bits ▷ `int` 32 Bits ▷ `long` 64 Bits
Binärdarstellung	▷ **Nicht**-negative Zahlen: ◇ Führendes Bit = **null** ▷ Negative Zahlen: ◇ Führendes Bit = **eins** ▷ Führendes Bit auch `Vorzeichenbit` genannt ▷ Größte darstellbare Zahl: $2^{N-1} - 1$ // Jedes Bit außer dem Ersten gesetzt
Umwandlung nicht-negativ in Dezimal	▷ Veranschaulichung am Datentyp `byte` ◇ `byte` maximal 127, deswegen reichen Zehnerpotenzen bis 100 ▷ Vorhergehensweise: (Beispiel 01101101 / 109) ◇ 01101101_2 / $01100100_2 = 00000001_2 \rightarrow$ "1??" - Ganzzahlige Division der Zahl durch 100 → 1 Rest 9 ◇ 01101101_2 % $01100100_2 = 00001001_2$ - Rest der vorherigen Division → 9 ◇ 00001001_2 / $00001010_2 = 00000000_2 \rightarrow$ "10?" - Teilen durch 2.höchste Zehnerpotenz (10^1) → 0 ◇ 00001001_2 % $00001010_2 = 00001010_2$ - Rest der vorherigen Division → 9 ◇ 00001001_2 / $00000001_2 = 00001001_2 \rightarrow$ "109" - Teilen durch kleinste Zehnerpotenz (10^0) → 9 - Teilen durch 1 natürlich überflüssig
Unicode-Kodierung	▷ Darstellung der Dezimalziffern: ◇ Setzen des 16er und 32er Bits auf 1 ◇ Also Addieren von 48 ◇ Bereich der Zahlenwerte: $48 - 57$
Umwandlung Dezimal in Datentyp	▷ Jede Zeichen der Zahl (z.B. 3856) stellt ja eine `char`-Zahl dar ◇ Subtrahieren von 48 (siehe Unicode Kodierung) ◇ Multiplikation mit dazugehöriger Zehnerpotenz ◇ Addieren der einzelnen Werte ▷ Schleife über Zehnerpotenzen, so oft wie die Zahl Ziffern enthält
Negative Zahlen (Zweierkomplement)	▷ Umwandlung von positiv nach negativ: ◇ Binärdarstellung der positiven Zahl ◇ Umdrehen aller Bits (1-Komplement) ◇ Addieren einer 1 (2-Komplement) ◇ Auch rückwärts anwendbar ▷ Zweierkomplement ermöglicht einfache Darstellung der 0 (0000) ▷ Zweierkomplement ermöglicht einfache Substraktion: ◇ Setzen des Subtrahenden ins Zweierkomplement ◇ Addieren der beide Werte ▷ Negativer Bereich ist um eins größer als Positiver

Bitlogik	▷ Überprüfung, ob Bit gesetzt ist: 1 `public static boolean bitIsSet(int bitArray, int position) {` 2 `  return bitArray & (1 « position) != 0;` 3 `}` ◇ `bitArray`: binäre Informationsquelle, 32 Bits ◇ `position`: auszulesende Stelle (31 MSB, 0 LSB) ◇ `1 « position`: Verschiebt Bitmuster um `position`-viele Stellen nach links - Schiebt 1 damit an abzufragende Stelle - `Linksshift`-Operator füllt alles rechts mit Nullen auf ◇ `&`: Bitweise `Verundung` (1, falls beide an der Stelle 1) - Ergibt an abzufragender Stelle genau 1, wenn `bitArray` an der Stelle auch 1 - Alle anderen Bits werden durch neues Bitmuster auf 1 gesetzt ◇ Am Ende Überprüfung mit `!= 0` ▷ Setzen eines einzelnen Bits: 1 `public static int setBit(int bitArray, int position) {` 2 `  return bitArray | (1 « position);` 3 `}` ◇ Selbes Verfahren wie bei der Überprüfung eines Bits ◇ Diesmal allerdings mit `|`: Bitweise Veroderung - Setzt das Bit an der gefragten Stelle immer auf 1 ▷ Nicht-Setzen eines einzelnen Bits: 1 `public static int unsetBit(int bitArray, int position) {` 2 `  return bitArray & ~(1 « position);` 3 `}` ◇ Selbes Verfahren wie bei der Überprüfung eines Bits ◇ Diesmal allerdings mit `~`: Komplement - Setzt das Bit immer auf 0 aufgrund des Komplements und der Verundung

Gebrochene Zahlen

Gebrochene Zahlen	▷ `float` 32 Bits ▷ `double` 64 Bits
Probleme mit Ungenauigkeiten	▷ Umkehrrechnungen liefern nicht genau den Ausgangswert ▷ Untergehen kleinerer Zahl bei Addition extrem unterschiedlich großer Zahlen ▷ Subtraktion fast gleich großer Zahlen führt mglw. zu inkorrekten Bits ▷ Ersetzen des Tests auf Gleichheit durch "ausreichend nahe beieinander"
Interne Darstellung	▷ $+3.14159E17$ ▷ `Vorzeichen`: Wird als binäre Information in einem einzelnen Bit abgespeichert ▷ `Basis`: Im Literal zur `Basis` 10, intern zur `Basis` 2 ▷ `Mantisse`: Die Gleitkommadarstellung der Zahl (3.14159) ▷ `Exponent`: Hier 17, Angabe der zu multiplizierende Potenz
IEEE 754	▷ Standard Nr. 754 der Vereinigung von Elektrotechnikern und Informatiker ◇ Regelt die binäre Darstellung von Gleitkommazahlen ▷ `Vorzeichen`: 1 Bit, 1 bedeutet negativ ▷ `Mantisse` und `Exponent` in normaler Binärdarstellung ◇ `float`: - `Mantisse:` 23 Bits - `Exponent:` 8 Bits ◇ `double`: - `Mantisse:` 52 Bits - `Exponent:` 11 Bits ◇ Ergibt mit dem einzelen Bit für `Vorzeichen` die Bitanzahl ▷ `Unendlich` und `NaN`: ◇ Auftreten des Falls: `Exponent` besteht nur aus Einsen ◇ `Mantisse` nur 0, dann `Unendlich` - Trotzdem vorzeichenabhängig ◇ Sonst als `NaN` (Not a Number)

28 Anhang: Korrekte Software

Korrektheit auf einzelnen Abstraktionsebenen

Lexikalische Ebene	▷ Darstellung typischer Fehler im Folgenden
	▷ Rechtschreibung
	▷ Formalisierung von Regeln:
	◇ Ähnliche Funktionsweise wie Grammatiken
	`identifier ::= «letter» «ident-char-list»`
	`ident-char-list ::= ϵ | «ident-char» «ident-char-list»`
	`ident-char ::= «letter» | «digit» | _ | $`
	`letter ::= a...z | A...Z`
	`digit ::= 0...9`
	◇ `::=` Formale Definition von Sprachkonstrukten
	◇ `Definiendum` links von `::=` (Name des Konstrukts)
	◇ `Definiens` rechts von `::=` (Definierende Ausdruck)
	◇ Verwendung von `«...»` bei Verwendung eines Konstrukts bei Definition
	◇ `|`: Trennt verschiedene Alternativen
	◇ ϵ steht für das leere Wort
	◇ Ableiten von korrekten `Identifiern` mithilfe dieser Grammatik

Syntaktische Ebene	▷ Definition Syntax:
	◇ Determiniert, ob ein Quelltext korrekt ist
	◇ Vorgegebene Regeln
	◇ Einfassen der kontextfreien Teile der Syntax in Regeln
	◇ d.h. Ignorieren aller Zusammenhänge des Quelltextes
	- z.B. ob Variable typgerechet verwendet wird
	▷ Syntaxfehler werden meist durch `Compiler` gefunden
	▷ Korrekte Klammersetzung:
	◇ Zu jeder öffnenden Klammer genau eine nachfolgende schließende Klammer
	◇ Zwei Klammerpaare immer nacheinander oder ineinander
	▷ Syntaktische Konstrukte:
	◇ Bildung anhand vorgegebener Struktur
	◇ z.B. `for`-Schleife: `for(..;..;..)`
	`statement ::= «compound-statement» | «if-statement» | ...`
	`compount-statement ::= { «statement-sequence» }`
	`statement-sequence ::= ϵ | «statement» «statement-sequence»`
	◇ Formale Definitionen erlauben Klärung von Detailfragen ("Darf ... leer sein?")
	`if-statement ::= if(«condition») «statement» |`
	`if(«condition») «statement» else «statement»`
	◇ Allgemein nützlich um die Syntax von Java nachzuvollziehen

Semantische Ebene	▷ Definition Semantik: ◇ Tatsächlicher Effekt eines sprach korrekten Programms ◇ ▷ Werden zur Laufzeit des Programms gefunden ◇ Werfen einer `RunTimeException` ▷ Beispiele: ◇ Teilen durch 0 ◇ Falscher `Array`-Index ◇ Zugriff auf `null`
Logische Ebene	▷ Umsetzungsfehler, Denkfehler ▷ Fehler bei der Übertragung von eigentlich richtigen Gedanken ▷ Beispiel: `off-by-one error` ◇ Richtige Berechnung, aber um 1 "daneben" ▷ Beispiel: Wochentag zu lang ◇ z.B.: Reservierung von acht Zeichen ◇ `Wednesday` jedoch neun Zeichen lang ▷ Logikfehler sind oft schwer zu finden

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

Korrektheit von Software

Korrektheit von Software	▷ Kein Programmabbruch durch Fehler ▷ Termination, wenn: ◇ Aufgabe erledigt ◇ Befehl zur Termination von außen ▷ Korrekte Ausgaben und Effekte
Korrektheit von Klassen	▷ Aufteilung in zwei Sammlungen von Aussagen: ◇ `Darstellungsinvariante` von Klassen und Interfaces - `representation invariant` - Beschreibt die Darstellung der Objekte gegenüber dem Nutzer der Klasse - Die Sicht, die Attribute und Methoden vermitteln, die `public` sind ◇ `Implementationsinvariante` von Klassen - `implementation invariant` - Analog zur Darstellungsinvariante - Behandelt den Teil der Klassendefinition, der nicht `public` ist - z.B.: Java-Kommentar in der Klassen-Quelldatei ▷ Umsetzungsbeispiele: ◇ Attribute `private` halten ◇ Zugriff auf Attribute nur über Methoden gewähren ◇ Überschreibung der geerbten Methode `clone()` und `equals` - Falls `equals` überschrieben wird, sollte auch `hashCode` überschrieben werden - Anforderungen an `equals` zu finden in Dokumentation `java.lang.Objekt` ▷ Formulierung Darstellungsinvariante Beispiel: Ein Objekt von Klasse `DMatrix` repräsentiert zu jedem Zeitpunkt seiner Lebenszeit eine Matrix von `double`; Zeilen- und Spaltenzahl sind konstant. - Beschreibung aller Begrenzungen, Umsetzungen, etc ▷ Formulierung Implementationsinvariante Beispiel: Attribut `matrix` vom Typ `double[][]` hat als Länge die Seitenzahl und seine Komponenten haben als Länge die Spaltenzahl. `matrix[i][j]` ist der Eintrag in Zeile `i` und Spalte `j`. - **ALLE** `private`-Attribute sollten hier angesprochen werden - Falls nicht, sind sie auch nicht wichtig genug überhaupt zu existieren - Parallele Entwicklung der Implementationsinvariante und dem Projekt ▷ Ableitung von Klassen / Implementationen von Interfaces ◇ Subtypen müssen Darstellungsinvariante einhalten - z.B. Methode in Subtyp muss selben Effekt auf Darstellungsinvariante haben - Liskov Substitution Principle ◇ Implementationsinvariante muss bei `protected`-Attributen der Basisklasse übernommen werden - `private`-Attribute nicht relevant, unter Kontrolle der Basisklasse ◇ Übernommen werden heißt: - Darf erweitert und verfeinert werden - **Nichts** darf zurückgenommen werden

Korrektheit von Subroutinen	▷ Subroutine als Oberbegriff für Methoden/Funktionen ▷ Vertrag zwischen dem Nutzer und dem Entwickler einer Subroutine ◇ Wenn der Aufruf alle Vorbedingungen erfüllt, muss die Subroutine alle Nachbedingungen erfüllen ◇ Vorbedingungen: - Implementationsinvariante vor dem Aufruf eingehalten - Parameter müssen gewisse Bedingungen erfüllen - Variable/Konstante außerhalb der Klasse - Externe Datenquellen (z.B. Dateien) ◇ Nachbedingungen: - Implementationsinvariante nach dem Aufruf eingehalten - Rückgabewert muss von bestimmtem Typ sein - Variable außerhalb der Klasse - Externe Datenquellen (z.B. Dateien) ▷ Aufbau des Vertrags: ◇ `Type` ◇ `Precondition` ◇ `Returns` ◇ `Postcondition` ▷ Ableitung von Basisklasse / Implementationen von Interface: ◇ Vorbedingung darf nur abgeschwächt werden, nicht verschärft oder ersetzt ◇ Nachbedingung darf nur verschärft werden, nicht abgeschwächt oder ersetzt ◇ Zweiter Teil des Liskov Substitution Principle
Korrektheit von rekursiven Subroutinen	▷ Rekursionsabbruch ◇ Muss vorhanden sein, damit Rekursion ordentlich terminiert ▷ Rekursionsschritt ◇ Schritt näher an den Rekursionsabbruch ▷ Beweis der Korrektheit mithilfe von Induktion: ◇ Induktionsbehauptung: Aufstellen für `Problemgröße` ◇ Induktionsanfang: z.B.: `Problemgröße` = 1 ◇ Induktionsvorraussetzung: Der Vertrag gelte für ... ◇ Induktionschritt: z.B.: Verringerung der Listenlänge
Korrektheit von Schleifen	▷ Schleifeninvariante : Aussagen darüber, was sich während Schleife nicht ändert ◇ Formulierung: "Nach h $\geq$ 0 Schritten ist ... " - Verwendung einer Variable (h), die nicht im Code vorkommt ▷ Schleifenvariante : Aussagen darüber, was sich während Schleife ändert ◇ Formulierung: `for`: "h steigt um 1 " ▷ Zusammenfassung: ◇ Formulierung: "Nach Schleifenende ist... " ▷ Induktion bei Schleifen: ◇ Invariante = Induktionsbehauptung: "Nach h $\geq$ 0 Schleifendurchläufen gilt..." ◇ Induktionsanfang, also h=0: "Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Durchlauf erfüllt ist." ◇ Induktionsvorraussetzung für h $>$ 0: "Die Invariante gelte für h-1." ◇ Induktionsschritt: "Unter Voraussetzung, dass..., nach h Durchläufen gilt."

29 Anhang: Effizienz von Software

Nebenaspekte der Effizienz

Speicherplatz	▷ Heutzutage z.B. auch bei größeren Datenmengen noch relevant ▷ Begriffe: ◇ Begrenzter Objektspeicher: `Heap` ◇ Voller Speicher: `OutOfMemoryError` ▷ Regeln: ◇ Vermeidung des unnötigen Festhaltens von Speicher ▷ Memory Leaks in `C` und `C++` ◇ Manuelle Freigabe von Speicherbereichen auf dem `Heap` notwendig ◇ Führt bei Nichtbeachtung zu Memory Leaks ▷ Speicherproblem bei Rekursion: ◇ Nur ein begrenzter Bereich für `Call-Stack` reserviert ◇ Zu tiefe Rekursion → `StackOverflowError`
Netzwerkbelastung	▷ Wird in späteren Veranstaltungen noch intensiver aufgegriffen ▷ Generisches Beispiel: Dezentrale Datenhaltung ◇ Daten werden redundant auf mehreren Netzwerkknoten gehalten - Mindestens ein Server, auf den Nutzer Zugriff hat, vorhanden ◇ Vorteile: - Je nach Belastungssituation, Weiterleitung zu anderem Server - Keine langen Wege im Netzwerk ◇ Nachteil: - Abgleich der Daten nach Änderung erfordert baldige Kommunikation ◇ Geeignete Wahl der Serverstruktur vonnöten
Reservierung von Ressourcen	▷ Ressourcen: Entitäten, die exklusiv von einem Prozess reserviert werden ▷ Wird später in Veranstaltungen zu Datenbanksystem näher behandelt ▷ z.B. Verwendung von `try-with-resources` ◇ Ressourcen werden automatisch in jedem Fall wieder geschlossen ◇ Ressourcen werden auf das zeitliche Minimum beschränkt
Pareto Regel	▷ Allgemein statistische Regel aus der VWL ▷ Übertragung auf das Thema effiziente Software: ◇ Nur wenige Quelltext is für den Großteil der Ineffizienz verantwortlich ▷ Konsequenz für Effizienzverbesserungen: ◇ 1. Prüfen wo die Effizienzverluste auftreten ◇ 2. Bei Verbesserungen auf diese Stellen konzentrieren

Hauptaspekt der Effizient - Laufzeit

Laufzeit messen

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

▷ Gewöhnliche Zeit Messung:

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

◇ auch noch Methode `nanoTime()`
- Jedoch nur Garantie, dass diese auf Millisekunden genau ist

▷ CPU-Zeit Messung:
◇ Messung im aufgerufenen Thread:

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

◇ Messung in anderen Threads:

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

- Zeile 2: Durchlauf durch alle Threads
- Zeile 3: Abfrage der verbrauchten CPU-Zeit eines Threads
- Zeile 4: Falls Thread schon terminiert, wird -1 zurückgeliefert

▷ Kommerzielle Tools zur Zeitmessung (Profiler):
◇ JVM Profiler:
- CPU-Zeit, Methodenaufrufe, Speichernutzung
- Abgriff dieser Daten direkt an der JVM
◇ Instrumentalisierende Profiler:
- Fügen weiteren Code zum Monitoring ein
- "instrumentalisieren" den zu überwachenden Code
- Jedoch zusätzliche Laufzeit
◇ Application Performance Monitoring (APM):
- instrumentialiserende Profile mit minimalistischer Datensammlung
- Geringer Laufzeit-Overhead
- Gesammelten Daten geben jedoch oft nur Hinweise
- Geeignet für Monitoring im produktiven Einsatz

Laufzeitverbesserungen

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

Asymptotische Komplexität

▷ Zur Vereinfachung: erst nur `User Time`
◇ Schätzung der Laufzeit durch eine mathematische Funktion
- In Kennzahlen, die die Problemgröße beschreiben
- Mathematische Überlegungen und/oder empirische Laufzeitstudien
- Es können auch mehrere Problemgrößen vorhanden sein
◇ Representative Operation Counts
- Identifikation der Anweisungen, die die Laufzeit dominieren könnten
- Ausführungen zählen für mathematische Überlegungen
- Laufzeiten akkumulieren bei Laufzeitstudien
- Akkumulation: Verwendung der Zeitmessung + Akkumulator

▷ Asymptotische Komplexität am Beispiel der linearen/binären Suche:
◇ Asymptotische Komplexität `(AK)` gibt an, in welcher Größenordnung die Laufzeit des Algorithmus mit der Problemgröße wächst
- `AK` von linearer Suche ist linear
- `AK` von binärer Suche ist logarithmisch

▷ `Worst Case` und `Best Case` am selben Beispiel
◇ Problemgröße hier: Zahl N der zu durchsuchenden Werte
- also die Länge des Arrays
◇ Für jede Problemgröße gibt es einen `Worst` und `Best Case`
- Zwei mathematische Funktionen in der Problemgröße
◇ `Best Case`:
- Fall, der die geringste Laufzeit produziert
- Hier: das erste angeschautete Element ist größer/gleich dem gesuchten
- Die Laufzeit im `Best` Case hängt **hier** nicht von N ab
◇ `Worst Case`:
- Fall, der die größte Laufzeit produziert
- Hier: Man muss die ganze Schleife durchlaufen
- Lineare Suche: N Durchläufe
- Binäre Suche: ca. $\log_2 N$ Durchläufe
◇ Bei großen Werte von N alle Operationen außer Schleife unerheblich
◇ Die Laufzeit pro Durchlauf variiert nur in engen Grenzen
◇ Die Laufzeiten pro Durchlauf bewegen sich in relativ engen Korridor
- $[c_1 * N \ ... \ c_2 * N]$ bei linearer Suche im `Worst Case`
- $[c_3 * \log_2 N \ ... \ c_4 * \log_2 N]$ bei binärer Suche im `Worst Case`
◇ Im `Best Case` bei beiden: $[c_5 \ ... \ c_6]$ **unabhängig von** N

▷ Schreibweise
◇ Seien $f : \mathbb{N} \to \mathbb{R}$ und $g : \mathbb{N} \to \mathbb{R}$
◇ Annahme: Es gibt beliebige, aber feste $c_u, c_o \in \mathbb{R}$ $(0 < c_u \leq c_o)$, so dass ab einer gewissen Größe der Eingabe n gilt:
$$c_u * g(n) \leq f(n) \leq c_o * g(n).$$
◇ Dann schreiben wir: $f \in \Theta(g)$.
- Θ: Menge aller Funktionen, die asymptotisch äquivalent zu g sind
- Korridor um die eine Funktion, die von der anderen nicht verlassen wird: asymptotisch gleich
◇ Bei einer konstanten Funktion g schreiben wir: $f \in \Theta(1)$
- Konstante Vergleichsfunktion, f bleibt in einem horizontalen Korridor
◇ Laufzeit bei linearer/binärer Suche:
- Lineare Suche im `Worst Case`: $\in \Theta(N)$
- Binäre Suche im `Worst Case`: $\in \Theta(\log_2 N)$
- Beide im `Best Case`: $\in \Theta(1)$

▷ Grenzen der Asymptotik
◇ Nicht sinnvoll, wenn es um kleine Problemgrößen geht
◇ Sehr viele kleine Probleme können aber trotzdem zu Laufzeitproblemen führen

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

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

30 Anhang: Fehlersuche und fehlervermeidender Entwurf

Fehlersuche

Fehlersuche	▷ Auftreten des Fehlers vs Ursache des Fehlers ◇ Stelle, an der Fehler auftritt nicht immer Stelle, an der Fehler passiert ist ◇ Zurückverfolgen der Fehlerursache ▷ Fehlerursache: ◇ Wo zum erstel Mal eine Vor-/Nachbedingung, In-/Variante nicht erfüllt ist ▷ Finden des Fehlers mithilfe von `Assert`-Anweisungen ▷ Wenn Fehlerursache nicht klar: ◇ Weitere `Assert`-Anweisungen, die weiter zurückgehen ◇ Temporäre `Assert`-Anweisungen für kritischen Datensatz - Hinzufügen eines bestimmten Kommentars für späteres Entfernen ▷ Fehler in Bibliothekskomponente eher auszuschließen ▷ Falls undurchschaubar: verdächtigen Quelltext neu implementieren
Laufzeittests	▷ Black-Box-Tests ◇ Testen einer Klasse von außen, ohne hineinzuschauen ◇ Verwenden von `JUnit`-Tests ◇ Jedoch auch spezifischere `Assert`-Anweisungen möglich ◇ Prüfung der Darstellungsinvariante bei Klassen/Interfaces ◇ Prüfung der Nachbedingung bei Subroutinen ▷ White-Box-Tests ◇ Hauptsächlich `Assert`-Anweisungen ◇ Implementationsinvarianten von Klassen ◇ Vor- und Nachbedingungen von Anweisungen ◇ Schleifen(in)varianten ▷ Vorhergehensweise: ◇ Testumgebung parallel zum eigentlichen Code entwickeln ◇ Änderung der Testumgebung falls Code-Änderung ◇ Beim Finden eines Fehlers: Einbau eines neuen Tests ▷ Coverage/Fehlerabdeckung: ◇ Randfälle: - z.B.: Dreiecke: Alle Ecken auf einer Linie ◇ Bei Verzweigungen: jeder mögliche Pfad: - Überprüfen jedes verschiedenen Ergebnisses ◇ n-verknüpfte `if`-Abfragen = 2^n Fälle zu testen

Fehlervermeidender Entwurf

Verbesserung	▷ Prinzipien und Techniken für fehlervermeidenden Entwurf verbessern auch: ◇ Wartbarkeit ◇ Modifizierbarkeit ◇ Erweiterbarkeit
KISS	▷ "keep it simple, stupid!" ▷ Dekomposition in kleine, überschaubare Einheiten (Klassen, Subroutinen, etc.) ▷ Programm muss nicht unbedingt besonders "raffiniert" sein ▷ Zerlegung in intuitiv sofort verständliche, realitätsabbildende Einheiten ▷ Gut gewählte Identifier für Einheiten ▷ Verwendung der für Java festgelegten Namenskonventionen ◇ Generelle Regel: - **Alle wichtigen** Aspekte zu Wortbestandteilen machen - Identifier ist der einzige Kommentar, der bei Nutzung dabei steht - Falls Identifier zu lang werden → Struktur gemäßKISS überdenken ◇ Ausnahmen: - Typische mathematische Notation (x,y,..) - Packagenamen eher kurz - Verwendung von allgemein bekannten Abkürzungen

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

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

Konformität

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

```
1  X[] a = new Y[200];
2  X x = a[150];
3  a[150] = new X();
```

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

31 Anhang: Polymorphie

Generelle Einteilung

Bisherige Konzepte	▷ `Racket`: ◇ Funktionen, die auf versch. Typen mit versch. Operationen arbeiten können - z.B. `foldr`, `map`, ... ▷ `Java`: ◇ Ableitungen von Klassen / Implementierung von Interfaces - Speicherung eines Subtyp-Objekts in Referenz des Basistyps - Typ in gewissen Grenzen frei wählbar ◇ Ad-hoc Polymorphie (eher primitiv) - Methodenüberladung und implizite Konversion ◇ `Generics` - Typparameter sind die polymorphen Typen
Einteilung von Konzepten	▷ Betrachung von zwei Zeitpunkten ◇ Zeitpunkt der Übersetzung ◇ Zeitpunkt an dem Ablauf an der Stelle im Quelltext ist ▷ Konformitätsprüfung: ◇ zur Kompilierzeit → statische Prüfung ◇ zur Laufzeit → dynamische Prüfung ▷ Ansteuerung der korrekten Implementation: (Auswahl der Subroutinen für Typ) ◇ zur Kompilierzeit → statische Bindung ◇ zur Laufzeit → dynamische Bindung ▷ Funktionen in `Racket`: ◇ Dynamische Prüfung (wird geprüft, wenn die Operation ausgeführt wird) ◇ Dynamische Bindung (Steuerung nach erfolgreicher Prüfung) ▷ `Java` - Klassen ableiten / Interfaces implementieren: ◇ Statische Prüfung (Inspektion des Quelltexts ausreichend) ◇ Dynamische Bindung (Auswahl welcher Implementation erst beim Aufruf) ▷ `Java` - Ad-hoc Polymorphie: ◇ Statische Prüfung ◇ Statische Bindung (Impliziten Konversionen) ◇ Dynamische Bindung (Methodenüberladung) - Hier wäre generell aber auch statische Bindung möglich (Java-spezifisch) ▷ `Java-Generics`: ◇ Statische Prüfung (Überprüfung ob Typparameter passend) ◇ Dynamische Bindung - Auch hier statische Bindung eigentlich möglich ▷ Typische Begriffe: ◇ **Duck-Typing**: dynamische Prüfung und Bindung ◇ **Subtyppolymorphie**: statische Prüfung und dynamische Bindung ◇ **Generizität**: statische Prüfung und Bindung

Abstraktion und Polymorphie

Abstraktion	▷ Chaos an Entitäten gedanklich in geeigneter Form strukturieren ◇ Zuordnung der Entitäten zu passenden Kategorien - `Differenzierung` ◇ Bei jeder Kategorie: - Herausfaktorisieren des Gemeinsamen aller Elemente in der Kategorie - können auch mehrere gemeinsame, zu trennende Aspekte sein - Unterschiedliches für sich stehen lassen - `Generalisierung` ◇ Beziehungen zwischen den Kategorien herstellen

Beispiel Racket	▷ Entitäten: Datenverarbeitungsaufgaben mittels einer Durchlauf durch Liste ◇ Selbe Standardaufgaben immer wieder ▷ Kategorie: `fold, filter` und `map` ▷ Bei jeder Kategorie: ◇ Gemeinsamkeit in jeweilige Funktion herausfaktorisiert ◇ Unterschiede: Unterschiedliche Parameter neben der Liste ▷ Beziehungen: `filter` und `map` sind **intermediate** ◇ `fold` ist hingegen **terminal**
Beispiel Java	▷ `Component: Button, Canvas,...` ◇ Gemeinsame Abstraktion: Arten von Komponenten in GUI ◇ Gemeinsamkeiten in `Component` herausfaktorisiert ▷ `Listener: KeyListener, MouseListener,...` ◇ Funktionalität der `Listener` herausfaktorisiert ◇ Getrennt von der jeweiligen Komponente ▷ `Event: ActionEvent, KeyEvent,...` ◇ Auch `Event` aus dem Konzept `Listener` ausgelagert
Abstraktion auf Typebene	▷ In logischer Einheit sind ein/mehrere Typen nicht festgelegt ▷ Können bei der Nutzung aus Menge von Typen gewählt werden ▷ Gewählte Typen müssen verlangte Funktionalitäten bieten ▷ Mehrene offene Typen müssen auch gemeinsam korrekt sein
Polymorphie	▷ Oberbegriff für alle Programmierkonzepte, mit denen Abstraktion auf Typebene realisiert werden kann ▷ Gründe für Polymorphie: ◇ Separation of Concerns - SoC hat aber natürlich auch viele weitere Gesichtspunkte ◇ Gleichbehandlung von Typen, wo die Unterschiede vernachlässigbar sind ◇ Einheit (deren Logik auf versch. Typen passt) nur einmal implementieren - Unterschiede dann in ausgelagerten Details
Konzepte von Konformität	▷ Konformität ist sehr vielfältig, viele Arten sie abzuprüfen ▷ Zusammenfassung bisheriger Stoff dazu: ◇ `Racket`: ob Operationen für Operanden definiert sind ◇ Suptyppolymorphie: nur Funktionalität des statischen Typs erlaubt ◇ Ad-hoc Polymorphie: - Methodenüberladung: Überprüfung, ob Signatur einzigartig - Implizite Konversion: Abprüfen eingebauter Regeln ◇ Generizität: unterschiedliche Modelle, sprachabhängig

Duck-Typing objektorientiert

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

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

◇ Zeile 3: Liefert das `Class`-Objekt für übergebene Klasse zurück
- Kompletter Name + Package-Pfad als Parameter
- `java.lang.Class` bietet Funktionalitäten rund um Klassen
- Zu jeder Klasse `X` existiert ein Objekt von `Class` (`Class<X>`)
- Existieren auch für primitive Datentypen und für `void`

◇ Zeile 4: Liefert die abgefragte Methode des `Class`-Objekts zurück
- Erste Parameter ist der Name der Methode
- beliebig viele weitere Parameter, methodenabhängig
- Muss genauso viele Parameter enthalten, wie die abgefragte Methode
- Jeder Parameter ist das `Class`-Objekt des Parametertyps

◇ Zeile 6: Aufruf der abgespeicherten Methode
- Parameter müssen natürlich übereinstimmen

◇ Duck-Typing Anpassung:
- Übergabe eines `Object obj` an Methode oben
- `Class<?> c = obj.getClass();` → **dynamische Prüfung**