

WESTFÄLISCHE HOCHSCHULE

OPR

ALLES WICHTIGE AUF EINEN BLICK

OPR Semesterzusammenfassung

L^AT_EX

Author(s):

-

Supervisor(s):

Prof. Luis

2. Juli 2025

This document, including appendices, is property of Westfälische Hochschule and is confidential, privileged and only for use of the intended addressees and may not be used, published or redistributed without the prior written consent of Westfälische Hochschule.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam a orci ornare nibh tincidunt molestie sed nec tellus. Morbi non sapien id lorem posuere pretium. Vestibulum commodo cursus purus, a elementum sem imperdiet sit amet. Phasellus posuere dolor dignissim aliquam tempus. Morbi egestas felis in lorem varius, ac egestas ante lacinia. Nulla sed ultrices dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Inhaltsverzeichnis

I	Vorlesungszusammenfassung	1
1	Pakete in Java	2
1.1	Importieren von Klassen und Paketen	2
1.1.1	Importieren aller Klassen eines Pakets (Wildcard-Import)	3
1.2	Das Paket <code>java.lang</code>	3
1.2.1	Statische Importanweisungen	4
2	Klassenhierarchie	5
2.1	Vererbung: Das Fundament der Hierarchie	5
2.1.1	Der Sichtbarkeitsmodifikator <code>protected</code>	6
2.1.2	Bindung von Methodenaufrufen: Statisch vs. Dynamisch	7
2.2	Das Schlüsselwort <code>super</code> (Prüfungsrelevant!)	9
2.3	Konstruktoraufrufe mit <code>this()</code>	12
2.4	Finale Klassen und Methoden	13
2.5	Abstrakte Klassen	14
3	JUnit Tests (Sehr Prüfungsrelevant)	16
3.1	Hintergrund von Tests	16
3.2	Tests schreiben mit JUnit	16
3.2.1	Importe für JUnit	16
3.2.2	Struktur einer Testklasse	17
3.2.3	Testdaten initialisieren mit <code>@BeforeEach</code>	18
3.2.4	Testdaten abbauen mit <code>@AfterEach</code> (seltener)	19
3.2.5	Testmethoden und Assertions	20
3.2.6	Mock-Klassen zum Testen abstrakter Klassen oder Abhängigkeiten	21
4	Klasse <code>Object</code>	23
4.0.1	<code>equals</code> und <code>hashCode</code>	23
4.0.2	<code>toString</code>	23
5	Collection-Klassen (Sehr Prüfungsrelevant)	24
5.1	Arrays	24
5.1.1	Anwendungsbereiche	24
5.1.2	Vorteile	24
5.1.3	Nachteile	24
5.1.4	Beispiel	24

5.2	ArrayList	25
5.2.1	Anwendungsbereiche	25
5.2.2	Vorteile	25
5.2.3	Nachteile	25
5.2.4	Beispiel	25
5.3	LinkedList	26
5.3.1	Anwendungsbereiche	26
5.3.2	Vorteile	26
5.3.3	Nachteile	26
5.3.4	Beispiel	26
5.3.5	Effizienz des Indexzugriffs bei LinkedLists (möglicherweise prüfungsrelevant)	27
5.4	HashSet	28
5.4.1	Anwendungsbereiche	28
5.4.2	Vorteile	28
5.4.3	Nachteile	28
5.4.4	Beispiel	28
5.5	TreeSet	29
5.5.1	Anwendungsbereiche	29
5.5.2	Vorteile	29
5.5.3	Nachteile	29
5.5.4	Beispiel	29
5.6	HashMap	30
5.6.1	Anwendungsbereiche	30
5.6.2	Vorteile	30
5.6.3	Nachteile	30
5.6.4	Beispiel	31
5.6.5	Prüfungshinweis	31
5.7	TreeMap	31
5.7.1	Anwendungsbereiche	31
5.7.2	Vorteile	32
5.7.3	Nachteile	32
5.7.4	Beispiel	32
5.8	Über Collections Iterieren (Prüfungshinweis)	33
6	Hüllenklassen	34
6.1	Definition und Notwendigkeit	34
6.2	Übersicht der Hüllenklassen	34
6.3	Erstellung von Hüllenklassen	34
6.4	Verwendung vor JDK 1.5	34
6.4.1	Ungültige Ausdrücke (Theoretisch)	35
6.4.2	Explizites Entpacken (Unboxing)	35
6.5	Autoboxing und Autounboxing (seit JDK 1.5)	35
6.6	Umwandlung von Strings in Hüllenklassenwerte	36
7	Exceptions (Ausnahmen)	37
7.1	Folgen einer Exception	37
7.2	Ausnahmebehandlung mit Try-Catch-Finally-Blöcken	38
7.2.1	Der try-Block	38

7.2.2	Der <code>catch</code> -Block	38
7.2.3	Der <code>finally</code> -Block (optional)	40
7.3	Unbehandelte Exceptions	40
8	Schnittstellen (Prüfungsrelevant)	42
8.1	Eigenschaften und Hinweise	42
8.2	Schnittstellen als Typ	42
8.3	Funktionale Schnittstellen	43
8.4	Lambda-Ausdrücke (sehr sehr Prüfungsrelevant)	43
8.4.1	Vereinfachung von Lambda-Ausdrücken	43
8.4.2	Hinweise zu Lambda-Ausdrücken	43
9	Streams	44
9.1	Beispiel	44
9.2	Arten von Streams	44
9.2.1	<code>IntStream</code> Beispiel	45
9.3	Hinweis	45
9.4	Stream erzeugen (Häufige Prüfungsaufgabe)	45
9.4.1	<code>Iterate</code>	45
9.4.2	im <code>LongStream</code>	46
9.4.3	<code>Generate</code>	46
9.4.4	<code>IntStream.range</code>	47
9.5	Operationen auf Streams	47
10	Ein- und Ausgabe	49
10.1	<code>InputStream</code>	49
10.1.1	<code>BufferedInputStream</code>	49
10.1.2	<code>ByteArrayInputStream</code>	50
10.2	<code>Reader</code>	50
10.2.1	<code>BufferedReader</code>	50
10.2.2	<code>StringReader</code>	50
10.3	Codierung	50
10.4	<code>OutputStream</code>	52
10.4.1	<code>BufferedOutputStream</code>	52
10.4.2	<code>ByteArrayOutputStream</code>	52
11	Aufzählungstypen	53
12	Collection-Klassen (Sehr Prüfungsrelevant)	54
12.1	Arrays	54
12.1.1	Anwendungsbereiche	54
12.1.2	Vorteile	54
12.1.3	Nachteile	54
12.1.4	Beispiel	54
12.2	<code>ArrayList</code>	55
12.2.1	Anwendungsbereiche	55
12.2.2	Vorteile	55
12.2.3	Nachteile	55
12.2.4	Beispiel	55

12.3	LinkedList	56
12.3.1	Anwendungsbereiche	56
12.3.2	Vorteile	56
12.3.3	Nachteile	56
12.3.4	Beispiel	56
12.3.5	Effizienz des Indexzugriffs bei LinkedLists (möglicher- weise prüfungsrelevant)	57
12.4	HashSet	58
12.4.1	Anwendungsbereiche	58
12.4.2	Vorteile	58
12.4.3	Nachteile	58
12.4.4	Beispiel	58
12.5	TreeSet	59
12.5.1	Anwendungsbereiche	59
12.5.2	Vorteile	59
12.5.3	Nachteile	59
12.5.4	Beispiel	59
12.6	HashMap	60
12.6.1	Anwendungsbereiche	60
12.6.2	Vorteile	60
12.6.3	Nachteile	60
12.6.4	Beispiel	61
12.6.5	Prüfungshinweis	61
12.7	TreeMap	61
12.7.1	Anwendungsbereiche	61
12.7.2	Vorteile	62
12.7.3	Nachteile	62
12.7.4	Beispiel	62
12.8	Über Collections Iterieren (Prüfungshinweis)	63
II	Übungsblätter	64
13	Übungsblatt Klassenhierarchie und Polymorphie	65
13.1	Aufgabe 1	65
13.2	Aufgabe 2	66
14	Übung Klasse Object	67
14.1	Aufgabe 1	67
14.2	Aufgabe 2	68
15	Übung Collection Klassen	71
15.1	Aufgabe 1	71
15.2	Aufgabe 2	72
15.3	Aufgabe 3	72
15.4	Aufgabe 4	73
15.5	Aufgabe 5	73
15.6	Aufgabe 6	74
16	Übung Ausnahmen	75

16.1 Aufgabe 1	75
16.2 Aufgabe 2	75
16.3 Aufgabe 3	76
17 Übung Schnittstellen	77
17.1 Aufgabe 1	77
17.2 Aufgabe 2	78
17.3 Aufgabe 3	79
17.4 Aufgabe 4	80
17.5 Aufgabe 5	80
18 Übung Funktionale Schnittstellen	81
18.1 Aufgabe 1	81
18.2 Aufgabe 2	81
19 Übung Streams	82
19.1 Aufgabe 1	82
19.2 Aufgabe 2	82
19.3 Aufgabe 3	83
19.4 Aufgabe 4	83
19.5 Aufgabe 5	83
19.6 Aufgabe 6	83
19.7 Aufgabe 7	83

Teil I

Vorlesungszusammenfassung

1. Pakete in Java

Pakete schaffen Ordnung. Sie bilden eine übersichtliche, hierarchische Struktur innerhalb von Java-Projekten. Im Prinzip sind Pakete Ordnerstrukturen, die Klassen und weitere (Unter-)Pakete enthalten können. Die Adressierung eines Paketes erfolgt durch seinen vollqualifizierten Paketpfad, der die Hierarchie widerspiegelt: $p_1.p_2.\dots.p_n$. Dabei ist p_n ein Paket innerhalb von p_{n-1} , welches wiederum in p_{n-2} liegt, und so weiter.

Klassen werden analog dazu über ihren vollqualifizierten Namen angesprochen: $p_1.p_2.\dots.p_n.K$, wobei K der Name der Klasse ist.

Listing 1.1: Beispiele für Paket- und Klassenspezifikationen

```
1 // Paket "regex", enthalten im Paket "util", das im ↵  
   Paket "java" liegt:  
2 java.util.regex  
3  
4 // Klasse "StringTokenizer", enthalten im Paket "util", ↵  
   das im Paket "java" liegt:  
5 java.util.StringTokenizer
```

Eine Klasse wird einem spezifischen Paket zugeordnet, indem die `package`-Anweisung als erste Zeile in der Quelldatei deklariert wird:

Listing 1.2: Deklaration der Paketzugehörigkeit

```
1 package java.util; // Beispielhafter Paketpfad  
2  
3 public class Main {  
4     // ...  
5 }
```

Diese Angabe ist für jede Java-Datei, die zu einem Paket gehört, obligatorisch. Vor der `package`-Anweisung dürfen lediglich Kommentare stehen - keine anderen Klassendefinitionen, Variablen oder Importanweisungen.

1.1 Importieren von Klassen und Paketen

Um Klassen aus anderen Paketen innerhalb der eigenen Klasse nutzen zu können, müssen diese *importiert* werden. Die `import`-Anweisungen stehen dabei stets am Anfang der Datei, nach der optionalen `package`-Deklaration und *vor* dem Klassenkopf (der Klassendefinition).

In modernen Entwicklungsumgebungen (IDEs) werden `import`-Anweisungen häufig automatisch hinzugefügt oder vorgeschlagen. Dennoch kann es in bestimmten Szenarien notwendig sein, diese manuell anzupassen oder zu ergänzen.

1.1.1 Importieren aller Klassen eines Pakets (Wildcard-Import)

Um alle öffentlichen Klassen eines bestimmten Pakets verfügbar zu machen, kann ein sogenannter Wildcard-Import verwendet werden. Dieser wird durch das Sternchen (*) symbolisiert. Beispielsweise importiert die Anweisung

Listing 1.3: Wildcard-Import des `java.util` Pakets

```
1 import java.util.*;
```

alle öffentlichen Klassen aus dem Paket `java.util`. Die importierten Klassen können daraufhin so verwendet werden, als wären sie im aktuellen Paket definiert worden.

Vorsicht bei Namenskonflikten: Wenn zwei oder mehrere Klassen mit identischem Namen aus unterschiedlichen Paketen importiert werden (oder eine importierte Klasse denselben Namen wie eine Klasse im aktuellen Paket hat), entsteht ein Namenskonflikt. In solchen Fällen müssen die betroffenen Klassen Vollständig Qualifiziert verwendet werden, um Eindeutigkeit zu gewährleisten. Beispiel:

Listing 1.4: Umgang mit Namenskonflikten

```
1 import java.util.Date;
2 import java.sql.Date;
3
4 public class DateTest {
5     java.util.Date utilDate; // Vollständig qualifiziert
6     java.sql.Date sqlDate;   // Vollständig qualifiziert
7
8     public void example() {
9         utilDate = new java.util.Date(); // Eindeutige ←
10             Zuweisung
11         // Date ambiguDate; // Fehler: Mehrdeutigkeit ohne ←
12             Qualifizierung
13     }
14 }
```

1.2 Das Paket `java.lang`

Das Paket `java.lang` nimmt eine Sonderstellung ein. Es enthält fundamentale Klassen, die für die grundlegende Funktionalität der Java-Sprache unerlässlich sind. Dazu gehören beispielsweise:

- `Object`: Die Wurzelklasse aller Java-Klassen.
- `String`: Für die Verarbeitung von Zeichenketten.

- System: Bietet Zugriff auf systemabhängige Ressourcen und Funktionen.
- Wrapper-Klassen für primitive Datentypen (z.B. Integer, Boolean).
- Math: Stellt mathematische Funktionen bereit.

Klassen aus dem `java.lang`-Paket sind **immer automatisch verfügbar** und müssen nicht explizit importiert werden.

1.2.1 Statische Importanweisungen

Neben dem Import von Klassen ist es auch möglich, statische Methoden und Variablen direkt zu importieren. Dies geschieht mithilfe der `import static`-Anweisung. Durch einen statischen Import können statische Mitglieder einer Klasse so aufgerufen werden, als wären sie in der aktuellen Klasse definiert, ohne den Klassennamen voranstellen zu müssen.

Beispiel für statischen Import: Die Anweisung

Listing 1.5: Statischer Import der max-Methode

```
1 import static java.lang.Math.max;
```

importiert die statische Methode `max` aus der Klasse `Math` (welche sich im Paket `java.lang` befindet). Anschließend kann die Methode direkt verwendet werden:

Listing 1.6: Verwendung nach statischem Import

```
1 public class MathExample {
2     public int biggerNumber(int a, int b) {
3         return max(a, b); // Aufruf ohne "Math." Präfix
4     }
5 }
```

Es ist auch möglich, alle statischen Mitglieder einer Klasse zu importieren:

Listing 1.7: Statischer Import aller Mitglieder von Math

```
1 import static java.lang.Math.*;
2
3 public class MathExampleAll {
4     public double circleSurfaceArea(double radius) {
5         return PI * pow(radius, 2); // PI und pow direkt ←
6         verfügbar
7     }
8 }
```

Wichtig: Nicht-statische Variablen oder Methoden können nicht über `import static` importiert werden. Diese Art des Imports ist ausschließlich statischen Mitgliedern vorbehalten. Instanzvariablen (nicht statische Variablen) können gar nicht importiert werden.

2. Klassenhierarchie

In der objektorientierten Programmierung ermöglicht die Klassenhierarchie die Schaffung von Beziehungen zwischen Klassen, wodurch Code-Wiederverwendung gefördert und eine logische Struktur aufgebaut wird. Dieses Kapitel erläutert die fundamentalen Konzepte der Vererbung, Sichtbarkeitsmodifikatoren und Bindungsarten in Java.

2.1 Vererbung: Das Fundament der Hierarchie

Eine Klasse kann Eigenschaften und Methoden von einer anderen Klasse übernehmen. Dieser Mechanismus wird als Vererbung bezeichnet und mit dem Schlüsselwort `extends` realisiert.

Syntax der Vererbung Die grundlegende Syntax zur Definition einer Klasse, die von einer anderen erbt, ist wie folgt:

Listing 2.1: Deklaration einer abgeleiteten Klasse

```
1 class Unterklasse extends Oberklasse {  
2     // Zusätzliche Attribute und Methoden der ←  
3     Unterklasse  
4     // oder überschriebene Methoden der Oberklasse  
5 }
```

In diesem Beispiel ist `Oberklasse` die direkte Superklasse (auch Elternklasse genannt) von `Unterklasse`. Umgekehrt ist `Unterklasse` eine direkte Subklasse (auch Kindklasse genannt) von `Oberklasse`. Eine Klasse kann beliebig viele direkte Unterklassen haben, jedoch stets nur **eine** direkte Oberklasse. Java unterstützt keine Mehrfachvererbung von Klassen. Die Kette der Oberklassen bildet die Vererbungshierarchie.

Was wird vererbt? Eine Unterklasse erbt von ihrer Oberklasse:

- Alle `public` und `protected` Instanzmethoden und Klassenmethoden (`static`).
- Alle Instanzvariablen und Klassenvariablen (`static`). Auch `private` deklarierte Variablen der Oberklasse sind Teil des Speicherabbaus von Objekten der Unterklasse. Jedoch ist ein direkter Zugriff auf `private` Variablen der Oberklasse aus der Unterklasse heraus nicht möglich; dieser kann nur über geerbte `public` oder `protected` Methoden der Oberklasse erfolgen (Prinzip der Datenkapselung).

Die Vererbung ist transitiv: Eine Klasse erbt auch die Member, die ihre Oberklasse bereits von deren Oberklasse geerbt hat. Geerbte Member verhalten sich so, als wären sie in der Unterklasse selbst deklariert, sofern sie nicht überschrieben werden.

Erweiterung und Spezialisierung durch die Unterklasse Unterklassen können die geerbten Funktionalitäten erweitern und spezialisieren:

- Sie können neue Methoden und Variablen definieren, die nur für die Unterklasse spezifisch sind.
- Sie können geerbte (nicht-final und nicht-private) Instanzmethoden **überschreiben** (`@Override`), um eine spezifische Implementierung bereitzustellen.

Nicht vererbte Elemente Folgende Elemente werden **nicht** an Unterklassen vererbt:

- **Konstruktoren:** Jede Klasse muss ihre eigenen Konstruktoren definieren oder den Default-Konstruktor verwenden. Der Konstruktor der Oberklasse kann jedoch mittels `super ()` aufgerufen werden (siehe Abschnitt 2.2).
- **Private Methoden:** Private Methoden der Oberklasse sind für die Unterklasse unsichtbar und werden daher nicht vererbt. Folglich können sie auch nicht im Sinne der Polymorphie überschrieben werden. Definiert eine Unterklasse eine Methode mit derselben Signatur wie eine private Methode der Oberklasse, so handelt es sich um eine vollkommen neue, unabhängige Methode. Es besteht keine polymorphe Beziehung zwischen diesen beiden Methoden; die Methode in der Unterklasse *verdeckt* (shadows) lediglich die Methode der Oberklasse, falls diese z.B. über einen Cast auf den Oberklassentyp aufgerufen würde (was bei privaten Methoden aber ohnehin nicht direkt von außen geht).

2.1.1 Der Sichtbarkeitsmodifikator **protected**

Der Modifikator `protected` stellt eine Sichtbarkeitsstufe zwischen `public` und `private` dar. Mit `protected` deklarierte Member (Methoden oder Variablen) sind sichtbar:

- Innerhalb der eigenen Klasse.
- Innerhalb aller Unterklassen, auch wenn diese sich in anderen Paketen befinden.
- Innerhalb aller anderen Klassen desselben Pakets.

Obwohl `protected` für Instanzvariablen in Vererbungsszenarien nützlich erscheinen mag, sollten Instanzvariablen aus Gründen der Kapselung und Wartbarkeit in den meisten Fällen als `private` deklariert werden. Der Zugriff sollte dann über `public` oder `protected` Getter- und Setter-Methoden erfolgen.

2.1.2 Bindung von Methodenaufrufen: Statisch vs. Dynamisch

Die Bindung legt fest, welche konkrete Methodenimplementierung bei einem Aufruf ausgeführt wird.

Statische Bindung (Early Binding) Bei der statischen Bindung wird bereits zur Kompilierzeit festgelegt, welche Methodenimplementierung ausgeführt wird. Dies geschieht typischerweise für:

- private-Methoden
- static-Methoden
- final-Instanzmethoden
- Aufrufe mit `super`.
- Aufrufe, bei denen der Compiler den exakten Objekttyp eindeutig bestimmen kann.

Listing 2.2: Beispiele für statische Bindung

```
1 class Vehicle {
2     public static String getCategorie() {
3         return "Allgemeines_Fahrzeug";
4     }
5     public final void startMotor() {
6         System.out.println("Motor_gestartet_(finale_Methode_ ↵
7             aus_Fahrzeug) ");
8     }
9 }
10 class Car extends Vehicle {
11     // Diese Methode verdeckt getCategorie() von Fahrzeug, ↵
12     // überschreibt sie aber nicht.
13     public static String getCategorie() {
14         return "Automobil";
15     }
16 }
17 public class TestBinding {
18     public static void main(String[] args) {
19         // Aufruf statischer Methoden: Immer statische ↵
20         // Bindung
21         System.out.println(Vehicle.getCategorie()); // Gibt ↵
22         // "Allgemeines Fahrzeug" aus
23         System.out.println(Car.getCategorie()); // Gibt ↵
24         // "Automobil" aus
25
26         Car meinAuto = new Car();
27         // Aufruf einer finalen Methode: Statische Bindung
28         meinAuto.startMotor(); // Ruft Vehicle.startMotor()
29
30         Vehicle meinFahrzeug = new Car();
31         // Obwohl meinFahrzeug auf ein Car-Objekt zeigt, ↵
32         // wird bei statischen Methoden
```

```

29 // der deklarierte Typ der Referenz (Car) für die ↵
    Bindung verwendet.
30 // System.out.println(meinFahrzeug.getCategory()); ↵
    // Schlechter Stil! Sollte Vehicle.getCategory() ↵
    sein.
31 // Würde "Allgemeines Fahrzeug" ausgeben.
32 }
33 }

```

Dynamische Bindung (Late Binding) Bei der dynamischen Bindung wird erst zur Laufzeit entschieden, welche Methodenimplementierung ausgeführt wird. Dies ist das Kernprinzip der Polymorphie und tritt bei überschriebenen Instanzmethoden auf. Der Java Virtual Machine (JVM) ermittelt den tatsächlichen Typ des Objekts, auf dem die Methode aufgerufen wird, und wählt die entsprechende Implementierung aus.

Listing 2.3: Beispiel für dynamische Bindung

```

1 class Animal {
2     public void say() {
3         System.out.println("Ein_Tier_gibt_einen_Laut_von_ ↵
            sich.");
4     }
5 }
6
7 class Dog extends Animal {
8     @Override
9     public void say() {
10        System.out.println("Wuff!_Wuff!");
11    }
12 }
13
14 class Cat extends Animal {
15     @Override
16     public void say() {
17        System.out.println("Miau!");
18    }
19 }
20
21 public class Zoo {
22     public static void main(String[] args) {
23         Animal[] tiere = new Animal[3];
24         tiere[0] = new Dog(); // Hund-Objekt
25         tiere[1] = new Cat(); // Katze-Objekt
26         tiere[2] = new Animal(); // Tier-Objekt
27
28         for (Tier aktuellesTier : tiere) {
29             // Dynamische Bindung:
30             // Zur Laufzeit wird die spezifische say()-Methode
31             // des tatsächlichen Objekttyps aufgerufen.
32             aktuellesTier.say();
33         }
34         // Ausgabe:
35         // Wuff! Wuff!

```



```

36     // Miau!
37     // Ein Tier gibt einen Laut von sich.
38 }
39 }

```

Der Compiler prüft lediglich, ob eine Methode `say()` im deklarierten Typ `Animal` existiert. Welche konkrete Implementierung dann ausgeführt wird, entscheidet die JVM basierend auf dem Laufzeittyp des Objekts in `aktuellesTier`.

2.2 Das Schlüsselwort **super** (Prüfungsrelevant!)

Das Schlüsselwort `super` hat in Java zwei primäre Verwendungszwecke im Kontext der Vererbung:

1. Aufruf des Konstruktors der direkten Oberklasse.
2. Zugriff auf Member (Methoden oder Variablen) der Oberklasse, die in der aktuellen Klasse möglicherweise überschrieben oder verdeckt wurden.

Aufruf des Oberklassen-Konstruktors mit `super()` Der Aufruf `super()` dient dazu, einen Konstruktor der direkten Oberklasse explizit auszuführen.

- Dieser Aufruf **muss**, falls vorhanden, die **erste Anweisung** im Konstruktor der Unterklasse sein.
- Die Parameterliste von `super(...)` muss mit der Signatur eines Konstruktors der Oberklasse übereinstimmen.

Listing 2.4: Expliziter Aufruf des Oberklassen-Konstruktors via `super()`

```

1  class BaseClass {
2      String name;
3      public BaseClass(String name) {
4          this.name = name;
5          System.out.println("Konstruktor_Basisklasse_ ←
6                               aufgerufen_mit:_" + name);
7      }
8      public BaseClass() {
9          this.name = "Default";
10         System.out.println("Parameterloser_Konstruktor_ ←
11                               Basisklasse_aufgerufen");
12     }
13 }
14
15 class ChildClass extends BaseClass {
16     public ChildClass(String spezifischerName) {
17         super(spezifischerName); // Ruft BaseClass(String ←
18                                   name) auf
19         System.out.println("Konstruktor_ChildClass_ ←
20                                   aufgerufen");
21     }
22     public ChildClass() {
23         super(); // Ruft BaseClass() auf
24         // Alternativ: super("DefaultAbgeleitet");
25     }
26 }

```

```

21     System.out.println("Parameterloser_Konstruktor_ ↵
        ChildClass_aufgerufen");
22 }
23 }
24
25 public class TestSuper {
26     public static void main(String[] args) {
27         ChildClass obj = new ChildClass("TestObjekt");
28         // Ausgabe:
29         // Konstruktor BaseClass aufgerufen mit: TestObjekt
30         // Konstruktor ChildClass aufgerufen
31
32         ChildClass obj2 = new ChildClass();
33         // Ausgabe:
34         // Parameterloser Konstruktor BaseClass aufgerufen
35         // Parameterloser Konstruktor ChildClass aufgerufen
36     }
37 }

```

Impliziter Aufruf von `super()` Wenn im Konstruktor einer Unterklasse **kein expliziter Aufruf** von `super(...)` (oder `this(...)`, siehe unten) als erste Anweisung steht, fügt der Java-Compiler automatisch einen parameterlosen Aufruf `super();` ein.

- Dies gilt auch für Klassen, die nicht explizit von einer anderen Klasse erben - diese erben implizit von der Klasse `Object`, die einen parameterlosen Konstruktor besitzt.
- Wenn eine Klasse gar keinen Konstruktor definiert, fügt der Compiler einen öffentlichen, parameterlosen Default-Konstruktor ein, der ebenfalls implizit `super();` aufruft.

Listing 2.5: Impliziter Konstruktor und `super()`-Aufruf durch den Compiler

```

1 class Ober {
2     public Ober() {
3         System.out.println("Konstruktor_Ober");
4     }
5 }
6
7 class Unter extends Ober {
8     // Kein expliziter Konstruktor definiert
9 }
10 // Der Compiler generiert für Klasse Unter:
11 // public Unter() {
12 //     super(); // Impliziter Aufruf des Ober-Konstruktors
13 // }
14
15 public class TestImplizit {
16     public static void main(String[] args) {
17         Unter u = new Unter(); // Führt zu Ausgabe: " ↵
            Konstruktor Ober"
18     }
19 }

```

Kompilierfehler bei fehlendem passenden Oberklassen-Konstruktor Der implizite `super();`-Aufruf (oder ein expliziter ohne Argumente) funktioniert nur, wenn die direkte Oberklasse einen zugänglichen, parameterlosen Konstruktor besitzt. Ist dies nicht der Fall (z.B. weil die Oberklasse nur Konstruktoren mit Parametern definiert), **muss** die Unterklasse in jedem ihrer Konstruktoren explizit einen passenden Konstruktor der Oberklasse mittels `super(...)` mit den erforderlichen Argumenten aufrufen. Andernfalls entsteht ein Kompilierfehler. Der Compiler kann die benötigten Argumente nicht erraten".

Listing 2.6: Kompilierfehler: `super()`-Aufruf scheitert bei parametrisiertem Superkonstruktor

```
1 class Elternteil {
2     private final String id;
3     public Elternteil(String id) { // Nur ein Konstruktor ←
        mit Parameter
4         this.id = id;
5     }
6 }
7
8 class KindValid extends Elternteil {
9     public KindValid(String kindId, String elternId) {
10         super(elternId); // Korrekt: Expliziter Aufruf ←
            des passenden Konstruktors
11     }
12 }
13
14 class KindError1 extends Elternteil {
15     public KindError1() {
16         // FEHLER! Implizites super() würde Elternteil() ←
            suchen, gibt es aber nicht.
17         // Explizites super("someID") wäre nötig.
18     }
19 }
20
21 class KindError2 extends Elternteil {
22     // FEHLER! Compiler fügt Default-Konstruktor ein, der ←
        super() aufruft.
23     // Elternteil() existiert nicht.
24 }
```

Zugriff auf Member der Oberklasse mit `super`. Mit `super.methodName()` oder `super.variablenName` kann auf eine Methode oder Variable der Oberklasse zugegriffen werden, selbst wenn diese in der aktuellen Klasse überschrieben (Methode) oder verdeckt (Variable) wurde. Dies ist nützlich, um die Funktionalität der Oberklasse zu erweitern, anstatt sie komplett zu ersetzen.

Listing 2.7: Zugriff auf überschriebene Methode der Oberklasse via `super`.

```
1 class Basis {
2     public void anzeige() {
3         System.out.println("Anzeige_aus_Basis");
```

```

4   }
5   protected String info = "Info_aus_Basis";
6 }
7
8 class Erweitert extends Basis {
9     @Override
10    public void anzeige() {
11        super.anzeige(); // Ruft anzeige() der Klasse Basis ←
        auf
12        System.out.println("Anzeige_aus_Erweitert");
13    }
14
15    public void zeigeInfos() {
16        System.out.println("Eigene_Info:_" + this.info); // ←
        this.info ist hier redundant, da info nicht neu ←
        deklariert wurde
17        System.out.println("Basis_Info:_" + super.info); // ←
        Greift auf info der Basisklasse zu
18    }
19
20    // Beispiel für Variablenverdeckung (Shadowing) - ←
    generell vermeiden!
21    // protected String info = "Info aus Erweitert";
22    // public void zeigeInfosMitVerdeckung() {
23    //     System.out.println("Eigene Info (verdeckt): " + ←
        this.info); // Info aus Erweitert
24    //     System.out.println("Basis Info (via super): " + ←
        super.info); // Info aus Basis
25    // }
26 }
27
28 public class TestSuperMember {
29     public static void main(String[] args) {
30         Erweitert ext = new Erweitert();
31         ext.anzeige();
32         // Ausgabe:
33         // Anzeige aus Basis
34         // Anzeige aus Erweitert
35         ext.zeigeInfos();
36     }
37 }

```

2.3 Konstruktoraufrufe mit **this()**

Analog zum Aufruf eines Oberklassen-Konstruktors mit `super()` kann mit `this()` ein anderer Konstruktor derselben Klasse aufgerufen werden. Dies wird als Konstruktorverkettung (constructor chaining) bezeichnet und dient dazu, Code-Duplizierung innerhalb verschiedener Konstruktoren einer Klasse zu vermeiden.

- Der Aufruf `this(...)` muss, falls vorhanden, die **erste Anweisung** im Konstruktor sein.

- Ein Konstruktor kann entweder einen `super(...)` oder einen `this(...)` Aufruf als erste Anweisung enthalten, aber nicht beide.
- Mindestens ein Konstruktor in der Kette muss (implizit oder explizit) den Konstruktor der Oberklasse via `super(...)` aufrufen.

Listing 2.8: Aufruf eines anderen Konstruktors derselben Klasse via `this()`

```

1  class Benutzer {
2      private final String benutzername;
3      private final String email;
4      private boolean istAktiv;
5
6      public Benutzer(String benutzername, String email) {
7          this.benutzername = benutzername;
8          this.email = email;
9          this.istAktiv = true; // Standardwert
10         System.out.println("Benutzer_erstellt:_" + ↵
11             benutzername + ",_Aktiv:_" + istAktiv);
12     }
13
14     public Benutzer(String benutzername) {
15         this(benutzername, benutzername + "@example.com"); ↵
16         // Ruft den ersten Konstruktor auf
17         System.out.println("Benutzer_(nur_Name)_erstellt,_E- ↵
18             Mail_generiert.");
19     }
20
21     public Benutzer() {
22         this("gast"); // Ruft den zweiten Konstruktor auf, ↵
23         der dann den ersten aufruft
24         this.istAktiv = false; // Überschreibt den ↵
25         Standardwert für Gast-Benutzer
26         System.out.println("Gast-Benutzer_erstellt,_Inaktiv_ ↵
27             gesetzt.");
28     }
29 }
30
31 public class TestThisKonstruktor {
32     public static void main(String[] args) {
33         Benutzer b1 = new Benutzer("alice", "alice@mail.com" ↵
34             );
35         System.out.println("---");
36         Benutzer b2 = new Benutzer("bob");
37         System.out.println("---");
38         Benutzer b3 = new Benutzer();
39     }
40 }

```

2.4 Finale Klassen und Methoden

Das Schlüsselwort `final` hat im Kontext der Vererbung spezielle Bedeutungen:

Finale Klassen Eine mit `final` deklarierte Klasse **kann nicht erweitert werden**, d.h., es können keine Unterklassen von ihr abgeleitet werden.

Listing 2.9: final Klasse

```
1 public final class Unveraenderlich {
2     // ...
3 }
4
5 // class VersuchAbleitung extends Unveraenderlich { } // ←
   KOMPILIERFEHLER!
```

Dies wird oft für Klassen verwendet, deren Implementierung als abgeschlossen und nicht für Erweiterungen vorgesehen gilt (z.B. `String` oder `Integer`).

Finale Methoden Eine mit `final` deklarierte Methode **kann in Unterklassen nicht überschrieben werden**.

Listing 2.10: final Methode

```
1 class BasisMitFinal {
2     public final void wichtigeOperation() {
3         System.out.println("Diese_Operation_darf_nicht_geä ←
4             ndert_werden.");
5     }
6     public void andereOperation() {
7         System.out.println("Diese_Operation_kann_ü ←
8             berschrieben_werden.");
9     }
10 }
11
12 class AbgeleitetVonFinal extends BasisMitFinal {
13     // @Override
14     // public final void wichtigeOperation() { } // ←
15     KOMPILIERFEHLER!
16
17     @Override
18     public void andereOperation() {
19         System.out.println("andereOperation_in_ ←
20             AbgeleitetVonFinal_überschrieben.");
21     }
22 }
```

Finale Methoden garantieren, dass das Verhalten dieser spezifischen Methode in der gesamten Klassenhierarchie unterhalb ihrer Definition konstant bleibt. Sie werden auch vom Compiler für Optimierungen herangezogen, da die Bindung statisch erfolgen kann.

2.5 Abstrakte Klassen

Abstrakte Klassen definieren ein Verhalten, das von anderen Klassen implementiert werden muss.

- Objekte abstrakter Klassen können nicht instanziiert werden.

- Methoden können in abstrakten Klassen deklariert werden, wobei deren Implementierung in ableitenden Klassen erforderlich ist.
- Bereits implementierte Methoden können ebenfalls in abstrakten Klassen enthalten sein, welche von abgeleiteten Klassen verwendet oder überschrieben werden können.

3. JUnit Tests (Sehr Prüfungs-relevant)

JUnit Tests sind ein fundamentaler Bestandteil der modernen Softwareentwicklung und ein häufiges Thema in Prüfungen. Dieses Kapitel führt in die Grundlagen des Testens mit JUnit ein.

3.1 Hintergrund von Tests

Software zu testen ist unerlässlich, um deren korrekte Funktionalität sicherzustellen. Eine bewährte Praxis ist das Test-Driven Development (TDD), bei dem Tests vor der eigentlichen Implementierung der Anwendungslogik geschrieben werden. Tests definieren die Anforderungen an das Programm und dienen als ausführbare Spezifikation. Testfälle repräsentieren dabei konkrete Anwendungsszenarien.

3.2 Tests schreiben mit JUnit

Testklassen in JUnit dienen dazu, die Methoden einer Anwendungsklasse systematisch zu überprüfen. Idealerweise existiert für jede Anwendungsklasse eine korrespondierende Testklasse. Diese Testklassen bündeln Testmethoden, die einzelne Aspekte oder Testfälle der zu testenden Methoden abdecken. Es ist üblich und empfehlenswert, mehrere Testmethoden für eine einzelne Anwendungsmethode zu erstellen, um verschiedene Szenarien (z.B. Normalfall, Grenzfälle, Fehlerfälle) abzudecken. Testmethoden sollten voneinander unabhängig sein und isoliert ausgeführt werden können.

3.2.1 Importe für JUnit

Für die Verwendung von JUnit müssen spezifische Annotationen und Assertions-Methoden importiert werden. JUnit ist ein weit verbreitetes Framework für das Testen von Java-Code.

Listing 3.1: Grundlegende JUnit 5 Importe

```
1 // Annotationen für Test-Setup und Testmethoden
2 import org.junit.jupiter.api.BeforeEach; // Wird vor ↔
   jeder Testmethode ausgeführt
3 import org.junit.jupiter.api.AfterEach; // Wird nach ↔
   jeder Testmethode ausgeführt (seltener benötigt)
```



```

4 import org.junit.jupiter.api.Test;           // Markiert ↵
   eine Methode als Testmethode
5 import org.junit.jupiter.api.Disabled;      // Deaktiviert ↵
   eine Testmethode oder -klasse
6
7 // Statische Importe für Assertions-Methoden (empfohlen ↵
   für bessere Lesbarkeit)
8 import static org.junit.jupiter.api.Assertions. ↵
   assertEquals;
9 import static org.junit.jupiter.api.Assertions. ↵
   assertTrue;
10 import static org.junit.jupiter.api.Assertions. ↵
   assertFalse;
11 import static org.junit.jupiter.api.Assertions. ↵
   assertNull;
12 import static org.junit.jupiter.api.Assertions. ↵
   assertNotNull;
13 import static org.junit.jupiter.api.Assertions. ↵
   assertThrows;
14 // Es gibt viele weitere Assertions-Methoden in org. ↵
   junit.jupiter.api.Assertions

```

Hinweis: Die Verwendung von *AfterEach* ist seltener notwendig als *BeforeEach*, da gut designte Tests oft keine explizite Aufräumarbeit nach jedem Test benötigen (z.B. wenn keine externen Ressourcen geöffnet werden).

3.2.2 Struktur einer Testklasse

Gemäß der Namenskonvention sollte eine Testklasse für eine Klasse 'MyClass' den Namen 'MyClassTest' tragen. Instanzvariablen in Testklassen sind üblich, um Testobjekte oder Testdaten zu halten, die von mehreren Testmethoden verwendet werden. Um die Unabhängigkeit der Tests zu gewährleisten, sollten diese Instanzvariablen typischerweise in einer mit '@BeforeEach' annotierten Methode 'setUp()' vor jeder Testmethode neu initialisiert werden.

Listing 3.2: Grundgerüst einer JUnit Testklasse

```

1 // Angenommen, wir testen eine Klasse namens "Calculator ↵
   "
2 class CalculatorTest {
3
4     private Calculator calculatorInstance; // ↵
       Instanzvariable für das Testobjekt
5
6     @BeforeEach
7     void setUp() {
8         // Diese Methode wird vor jeder @Test Methode ausgef ↵
          ührt.
9         // Ideal für die Initialisierung von Testobjekten.
10        calculatorInstance = new Calculator();
11        System.out.println("setUp()_executed:_New_Calculator ↵
           _instance_created.");
12    }
13

```

```

14  @AfterEach
15  void tearDown() {
16      // Diese Methode wird nach jeder @Test Methode ↵
        ausgeführt.
17      // Nützlich für Aufräumarbeiten, z.B. Schließen von ↵
        Ressourcen.
18      // In vielen Fällen nicht zwingend notwendig.
19      calculatorInstance = null; // Beispiel: Objekt ↵
        freigeben
20      System.out.println("tearDown()_executed.");
21  }
22
23  @Test
24  void testAddition() {
25      // Testlogik für die Additionsmethode
26      int result = calculatorInstance.add(5, 3);
27      assertEquals(8, result, "5_+_3_should_be_8"); // ↵
        Assertion mit optionaler Nachricht
28  }
29
30  @Test
31  void testSubtraction() {
32      // Testlogik für die Subtraktionsmethode
33      int result = calculatorInstance.subtract(10, 4);
34      assertEquals(6, result, "10_-_4_should_be_6");
35  }
36
37  @Test
38  @Disabled("This_test_is_currently_disabled_due_to_ ↵
        ongoing_refactoring.")
39  void testMultiplication() {
40      // Ein deaktivierter Test
41      assertEquals(12, calculatorInstance.multiply(3,4));
42  }
43  }
44
45  // Dummy-Klasse, die getestet wird (normalerweise in ↵
        einer separaten Datei)
46  class Calculator {
47      public int add(int a, int b) { return a + b; }
48      public int subtract(int a, int b) { return a - b; }
49      public int multiply(int a, int b) { return a * b; }
50  }

```

3.2.3 Testdaten initialisieren mit @BeforeEach

Um sicherzustellen, dass jede Testmethode mit einem sauberen und definierten Zustand startet, werden Testdaten und -objekte häufig in einer Methode initialisiert, die mit '@BeforeEach' annotiert ist. Die Namenskonvention für diese Methode ist 'setUp()'.

Listing 3.3: Verwendung von @BeforeEach zur Initialisierung

```

1  class Book { // Beispielklasse, die getestet wird
2      private String title;

```

```

3   private String author;
4   // Weitere Attribute und Konstruktor...
5   public Book(String title, String author) { this.title ←
      = title; this.author = author; }
6   public String getAsText() { return title + ";_ " + ←
      author; }
7 }
8
9 class BookTest {
10  private Book testBook; // Instanzvariable für das ←
      Testobjekt
11
12  @BeforeEach
13  void setUp() {
14      // Initialisiert das testBook Objekt vor jedem Test ←
      neu
15      testBook = new Book("The_Hitchhiker's_Guide_to_the_ ←
      Galaxy", "Douglas_Adams");
16  }
17
18  @Test
19  void testBookCreation() {
20      assertNotNull(testBook, "Book_should_be_initialized_ ←
      by_setUp()");
21  }
22
23  @Test
24  void testGetAsText() {
25      String expectedText = "The_Hitchhiker's_Guide_to_the_ ←
      _Galaxy;_Douglas_Adams";
26      assertEquals(expectedText, testBook.getAsText(), " ←
      getAsText()_should_return_correct_format.");
27  }
28 }

```

3.2.4 Testdaten abbauen mit @AfterEach (seltener)

Falls nach der Ausführung jeder Testmethode Aufräumarbeiten notwendig sind (z.B. das Schließen von Dateien, Netzwerkverbindungen oder das Zurücksetzen von globalen Zuständen), kann dies in einer mit '@AfterEach' annotierten Methode geschehen. Die Namenskonvention hierfür ist 'tearDown()'. Für die meisten Unit-Tests, die mit einfachen Objekten arbeiten, ist dies nicht erforderlich.

Listing 3.4: Verwendung von @AfterEach für Aufräumarbeiten

```

1 // @AfterEach // Selten benötigt für einfache Unit-Tests
2 // void tearDown() {
3 //     // Diese Methode wird nach jedem Testfall ←
      aufrufen.
4 //     // Beispiel: Schließen einer Datei, Freigeben von ←
      Ressourcen.
5 //     // System.out.println("tearDown() called.");
6 // }

```

3.2.5 Testmethoden und Assertions

Testmethoden werden mit `@Test` annotiert. Die Namenskonvention für Testmethoden ist `test<MethodenNameDieGetestetWird>[_Szenario]`, z.B. `testAdd_withPositiveNumbers` oder `testAdd_withNegativeNumbers`. Testmethoden haben in der Regel keine Rückgabewerte (`void`) und keine Modifikatoren (package-private Sichtbarkeit ist üblich). Sie sollten prägnant sein und sich auf die Überprüfung eines spezifischen Aspekts konzentrieren.

Innerhalb einer Testmethode werden Assertions verwendet, um das tatsächliche Verhalten des Codes mit dem erwarteten Verhalten zu vergleichen.

Listing 3.5: Beispiel einer Testmethode mit Assertions

```
1 // Innerhalb einer Testklasse, z.B. BookTest (siehe oben ↵
  )
2
3 // @BeforeEach
4 // void setUp() {
5 //     testBook = new Book("Asterix der Gallier", "Uderzo ↵
      ", 1965, 9.8);
6 // }
7
8 // Angenommen, die Book-Klasse hat eine Methode ↵
    getFormattedString():
9 // public String getFormattedString() {
10 //     return String.format("%s; %s; %d; %.1f", title, ↵
        author, year, price);
11 // }
12 // Und der Konstruktor ist: public Book(String title, ↵
    String author, int year, double price)
13
14 // @Test
15 // void testGetFormattedString_validBook() {
16 //     // Annahme: testBook wurde in setUp() initialisiert
17 //     // testBook = new Book("Asterix der Gallier", " ↵
        Uderzo", 1965, 9.8); // Falls kein setUp
18 //     String expectedOutput = "Asterix der Gallier; ↵
        Uderzo; 1965; 9.8";
19 //     // assertEquals(expectedOutput, testBook. ↵
        getFormattedString());
20 // }
```

In diesem Beispiel würde `assertEquals` prüfen, ob der von `book.getFormattedString()` zurückgegebene String dem `expectedOutput` entspricht. Die `equals`-Methode des Vergleichsobjekts wird hierfür herangezogen (für Strings ist dies ein Inhaltsvergleich).

Einige gängige Assertions sind:

- `assertEquals(expected, actual, [message])`: Überprüft, ob zwei Werte gleich sind.
- `assertTrue(condition, [message])`: Überprüft, ob eine Bedingung wahr ist.

- `assertFalse(condition, [message])`: Überprüft, ob eine Bedingung falsch ist.
- `assertNull(object, [message])`: Überprüft, ob ein Objekt null ist.
- `assertNotNull(object, [message])`: Überprüft, ob ein Objekt nicht null ist.
- `assertArrayEquals(expectedArray, actualArray, [message])`: Vergleicht zwei Arrays auf Gleichheit (Element für Element).
- `assertThrows(expectedThrowable, executable, [message])`: Überprüft, ob die Ausführung des 'executable' Codes eine Exception vom Typ 'expectedThrowable' wirft.

Die optionale 'message' wird angezeigt, wenn die Assertion fehlschlägt, was die Fehlersuche erleichtert.

3.2.6 Mock-Klassen zum Testen abstrakter Klassen oder Abhängigkeiten

Um abstrakte Klassen zu testen oder Klassen, die komplexe Abhängigkeiten haben, werden oft Mock-Klassen (oder Mock-Objekte) verwendet. Eine Mock-Klasse ist eine vereinfachte Implementierung einer Abhängigkeit, die speziell für Testzwecke erstellt wird. Sie simuliert das Verhalten der echten Abhängigkeit in einer kontrollierten Weise. Die Namenskonvention für eine Mock-Klasse für 'AbstractDataProvider' könnte 'MockAbstractDataProvider' oder 'TestDataProviderImpl' sein.

Listing 3.6: Konzept einer Mock-Klasse (vereinfacht)

```

1 // Abstrakte Klasse, die getestet werden soll (indirekt)
2 abstract class AbstractDataProcessor {
3     protected abstract String fetchData(); // Abhängigkeit ←
4     , die gemockt werden könnte
5
6     public String processData() {
7         String data = fetchData();
8         return "Processed:_" + data.toUpperCase();
9     }
10 }
11 // Eine konkrete Implementierung der abstrakten Klasse f ←
12 // für Testzwecke (eine Art Mock)
13 class TestableDataProcessor extends AbstractDataProcessor {
14     private String dataToReturn;
15
16     public TestableDataProcessor(String dataToReturn) {
17         this.dataToReturn = dataToReturn;
18     }
19
20     @Override
21     protected String fetchData() {

```

```

21     // Simuliert das Verhalten der Abhängigkeit
22     return this.dataToReturn;
23 }
24 }
25
26 // Testklasse
27 class AbstractDataProcessorTest {
28     @Test
29     void testProcessData_withMockedData() {
30         // Erstellt eine Instanz der Test-Implementierung ↵
31         // mit kontrollierten Daten
32         AbstractDataProcessor processor = new ↵
33             TestableDataProcessor("sample_data");
34         String result = processor.processData();
35         assertEquals("Processed: SAMPLE_DATA", result);
36     }
37 }

```

Für komplexere Mocking-Szenarien werden oft Mocking-Frameworks wie Mockito oder EasyMock eingesetzt, die das Erstellen und Konfigurieren von Mock-Objekten erheblich vereinfachen. Diese sind jedoch über den Rahmen dieser Einführung hinausgehend.

4. Klasse Object

Die Klasse Object ist eine besondere Klasse. *jede* Klasse erbt von der Object Klasse. In der Object Klasse sind einige Methoden definiert, welche jede Klasse besitzt. Zu ihnen gehören beispielsweise `equals` und `hashCode`

4.0.1 equals und hashCode

Die `equals` und `hashCode` Methoden haben einen Vertrag. Wenn eine von beiden überschrieben wird, muss die andere auch überschrieben werden. Wenn Objekte im Sinne von `equals` gleich sind, *müssen* diese auch den selben `hashCode` besitzen. Wenn Objekte jedoch den selben `hashCode` haben, müssen sie nicht zwingend im Sinne von `equals` gleich sein.

Standardmäßig prüft die `equals` Methode auf Gleichheit der Identität, die `hashCode` Methode gibt die Memory Adresse zurück.

4.0.2 toString

Das Objekt wird textuell ausgegeben. Diese Methode wird beispielsweise aufgerufen, wenn ein Objekt über ein `System.out.println` in die Konsole ausgegeben wird.

Die Klasse Object hat noch viele weitere Methoden implementiert, allerdings sind keine weiteren Prüfungsrelevant.

5. Collection-Klassen (Sehr Prüfungsrelevant)

Arrays haben eine feste Größe, die sich während der Laufzeit nicht ändern kann. Dies stellt häufig eine Einschränkung dar. Sogenannte Collection-Klassen (oder Kollektionen) bieten hier flexiblere Alternativen. Es gibt verschiedene Arten von Collection-Klassen, die sich grob in Listen (`List`), Mengen (`Set`) und Abbildungen (`Map`) einteilen lassen. Jede Art und ihre spezifischen Implementierungen haben eigene Vor- und Nachteile, die sie für unterschiedliche Anwendungsfälle geeignet machen.

5.1 Arrays

Arrays sind streng genommen keine Collection-Klassen im Sinne des Java Collections Frameworks. Dennoch ist es sinnvoll, sie hier zu betrachten, da sie einen grundlegenden Mechanismus zur Gruppierung von Elementen darstellen und als Vergleichsbasis für die eigentlichen Collection-Klassen dienen.

5.1.1 Anwendungsbereiche

Arrays eignen sich, wenn eine feste Anzahl von Elementen desselben Typs geordnet gespeichert werden soll und sich diese Anzahl zur Laufzeit nicht ändert.

5.1.2 Vorteile

Der Zugriff auf Elemente über ihren Index ist bei Arrays sehr schnell und effizient (konstante Zeitkomplexität, $O(1)$).

5.1.3 Nachteile

Die Größe eines Arrays ist nach seiner Initialisierung unveränderlich. Eine nachträgliche Größenänderung erfordert die Erstellung eines neuen Arrays und das Kopieren der Elemente.

5.1.4 Beispiel

Listing 5.1: Beispiel für die Verwendung eines Arrays in Java

```
1  int[] numbers = new int[5]; // Array der Größe 5 ←  
    erstellen
```



```

2 numbers[3] = 42; // Dem Element am Index 3 den Wert 42 ↔
   zuweisen
3 System.out.println(numbers[3]); // Wert am Index 3 ↔
   ausgeben (Ausgabe: 42)
4 // numbers[5] = 7; // Dies würde eine ↔
   ArrayIndexOutOfBoundsException auslösen

```

5.2 ArrayList

ArrayLists sind dynamische Arrays, d.h., sie können ihre Größe während der Laufzeit anpassen. Sie implementieren das List-Interface.

5.2.1 Anwendungsbereiche

ArrayLists sind dann sinnvoll, wenn Elemente geordnet gespeichert werden sollen und sich die Anzahl der Elemente dynamisch ändern kann. Sie bieten einen guten Kompromiss zwischen flexiblem Größenmanagement und schnellem Indexzugriff.

5.2.2 Vorteile

- Die Größe der Liste kann zur Laufzeit dynamisch wachsen oder schrumpfen.
- Der Zugriff auf Elemente über ihren Index ist in der Regel schnell (amortisiert $O(1)$), ähnlich wie bei Arrays.

5.2.3 Nachteile

- Das Einfügen oder Löschen von Elementen am Anfang oder in der Mitte der Liste kann langsam sein ($O(n)$), da nachfolgende Elemente verschoben werden müssen.
- Beim Überschreiten der internen Kapazität muss das zugrundeliegende Array vergrößert und die Elemente kopiert werden. Obwohl das Hinzufügen am Ende amortisiert $O(1)$ ist, kann diese Operation vereinzelt länger dauern.

5.2.4 Beispiel

Listing 5.2: Beispiel für die Verwendung einer ArrayList in Java

```

1 import java.util.ArrayList; // Import notwendig
2 import java.util.List; // Interface verwenden ist gute ↔
   Praxis
3
4 List<Integer> list = new ArrayList<>(); // Neue, leere ↔
   ArrayList erstellen
5 list.add(10); // Element 10 am Ende hinzufügen
6 list.add(20); // Element 20 am Ende hinzufügen
7 list.add(0, 5); // Element 5 am Index 0 einfügen ( ↔
   verschiebt andere Elemente)
8

```

```

9 | System.out.println(list.get(1)); // Element am Index 1  ↔
   |     ausgeben (Ausgabe: 10)
10 | list.remove(0); // Element am Index 0 entfernen

```

5.3 LinkedList

Eine `LinkedList` speichert ihre Elemente in einer doppelt verketteten Liste. Jeder Knoten enthält das Element selbst sowie Verweise auf das vorherige und das nächste Element in der Sequenz. Sie implementiert ebenfalls das `List`-Interface sowie das `Deque`-Interface (Double-Ended Queue), siehe `Deque` (Double-Ended Queue).

5.3.1 Anwendungsbereiche

`LinkedLists` eignen sich besonders gut, wenn häufig Elemente am Anfang oder Ende der Liste hinzugefügt oder entfernt werden müssen. Sie sind daher eine gute Wahl für die Implementierung von Stacks (Stapelspeicher (Stack), LIFO (Last-In, First-Out)) oder Queues (Warteschlangen, FIFO (First-In, First-Out)).

5.3.2 Vorteile

- Sehr schnelles Einfügen und Löschen von Elementen am Anfang und Ende der Liste ($O(1)$).
- Dynamische Größenanpassung ohne die Notwendigkeit, große Speicherblöcke am Stück zu reservieren oder umzukopieren.

5.3.3 Nachteile

- Der Zugriff auf ein Element anhand seines Indexes (z.B. mit `get(index)`) ist langsam ($O(n)$), da die Liste vom Anfang (oder Ende, je nachdem, was näher ist) bis zum gewünschten Element durchlaufen werden muss.
- Benötigt mehr Speicher pro Element als eine `ArrayList`, da für jeden Knoten zusätzliche Referenzen auf Vorgänger und Nachfolger gespeichert werden müssen.

5.3.4 Beispiel

Listing 5.3: Beispiel für die Verwendung einer `LinkedList` in Java

```

1 | import java.util.LinkedList; // Import notwendig
2 | import java.util.List;      // Interface verwenden
3 |
4 | List<String> queueList = new LinkedList<>(); // Neue,  ↔
   |     leere LinkedList erstellen
5 | // Für Queue/Deque-Operationen kann man auch direkt  ↔
   |     LinkedList oder Deque verwenden:
6 | // Deque<String> queue = new LinkedList<>();
7 |
8 | queueList.add("Erster"); // Element am Ende anfügen

```

```

9  ((LinkedList<String>) queueList).addFirst("Neuer_Erster" ←
    ); // Spezifische LinkedList-Methode
10 ((LinkedList<String>) queueList).offer("Letzter"); // ←
    Typisch für Queues (fügt am Ende hinzu)
11
12 System.out.println(queueList.get(1)); // Element am ←
    Index 1 ausgeben (Ausgabe: "Erster")
13 System.out.println(((LinkedList<String>) queueList).poll ←
    ()); // Erstes Element abrufen und entfernen (Ausgabe ←
    : "Neuer Erster")

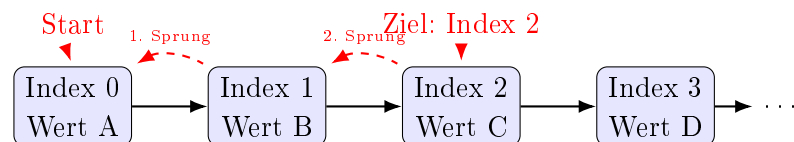
```

5.3.5 Effizienz des Indexzugriffs bei LinkedLists (möglicherweise prüfungsrelevant)

Der Zugriff auf ein Element an einem bestimmten Index (z.B. mittels `liste.get(index)`) ist bei einer `LinkedList` vergleichsweise langsam und hat eine Zeitkomplexität von $O(n)$ im schlechtesten und durchschnittlichen Fall. Der Grund hierfür liegt in der Struktur der verketteten Liste:

- Anders als bei einem Array, wo die Speicheradresse eines Elements direkt aus dem Index berechnet werden kann, muss bei einer `LinkedList` die Kette von Verweisen vom ersten (oder letzten, je nach Implementierung und Index) Knoten bis zum gewünschten Knoten verfolgt werden.
- Jeder Knoten kennt nur seinen direkten Vorgänger und Nachfolger. Um zum i -ten Element zu gelangen, müssen i Schritte (oder $n-i$ Schritte vom Ende) durchlaufen werden.

Die nachfolgende Abbildung 12.1 verdeutlicht diesen Prozess.



Zugriff auf Index 2: Es müssen 2 Sprünge vom Start (Index 0) erfolgen, um das Element zu erreichen. Jeder Sprung folgt dem Verweis des aktuellen Knotens auf den nächsten Knoten.

Abbildung 5.1: Visualisierung des sequenziellen Zugriffs auf ein Element (hier Index 2) in einer `LinkedList`. Um zum Zielknoten zu gelangen, muss die Liste vom Anfangsknoten aus durchlaufen werden.

Im Gegensatz dazu ist das sequentielle Durchlaufen aller Elemente einer `LinkedList` mit einem Iterator (z.B. in einer `for-each`-Schleife) effizient. Jeder Schritt des Iterators zum nächsten Element dauert $O(1)$, sodass das Iterieren über die gesamte Liste eine Gesamtkomplexität von $O(n)$ hat. Problematisch ist also nicht das Iterieren an sich, sondern der wahlfreie Zugriff

per Index.

5.4 HashSet

Ein `HashSet` implementiert das `Set`-Interface und speichert eine Sammlung von eindeutigen Elementen. Die Elemente in einem `HashSet` haben keine garantierte Reihenfolge; die Iterationsreihenfolge kann sich sogar ändern, wenn neue Elemente hinzugefügt werden. Die Implementierung basiert auf einer Hashtabelle (intern wird oft eine `HashMap` verwendet, bei der die Werte ignoriert werden).

5.4.1 Anwendungsbereiche

`HashSets` eignen sich hervorragend, wenn:

- Eindeutige Elemente gespeichert werden sollen und die Reihenfolge keine Rolle spielt.
- Schnell geprüft werden muss, ob ein Element bereits in der Sammlung vorhanden ist.
- Duplikate automatisch verhindert werden sollen.

5.4.2 Vorteile

- Sehr schnelle durchschnittliche Zeitkomplexität von $O(1)$ (amortisiert) für die Operationen `add`, `remove` und `contains`.
- Effiziente Vermeidung von Duplikaten.

5.4.3 Nachteile

- Keine garantierte Reihenfolge der Elemente. Die Reihenfolge bei der Iteration ist nicht vorhersagbar.
- Die Leistung kann bei einer schlechten Implementierung der `hashCode()`-Methode der gespeicherten Objekte oder bei einer sehr hohen Füllrate der internen Hashtabelle (viele Kollisionen) im schlechtesten Fall auf $O(n)$ für einzelne Operationen abfallen.

5.4.4 Beispiel

Listing 5.4: Beispiel für die Verwendung eines `HashSet` in Java

```
1 import java.util.HashSet;
2 import java.util.Set; // Interface verwenden
3
4 Set<String> uniqueNames = new HashSet<>();
5 uniqueNames.add("Alice");
6 uniqueNames.add("Bob");
7 uniqueNames.add("Alice"); // Wird ignoriert, da Duplikat ←
   (add() gibt false zurück)
8
9 System.out.println(uniqueNames.contains("Alice")); // ←
   true
```

```

10 System.out.println(uniqueNames.size()); // 2
11
12 for (String name : uniqueNames) {
13     System.out.println(name); // Reihenfolge nicht ↔
14     // garantiert, z.B. Bob, Alice
15 }

```

5.5 TreeSet

Ein `TreeSet` implementiert das `SortedSet`-Interface (welches `Set` erweitert) und speichert eine Sammlung von eindeutigen Elementen in sortierter Reihenfolge. Die Sortierung erfolgt entweder natürlich (wenn die Elemente `Comparable` implementieren) oder durch einen beim Erstellen des `TreeSet`s übergebenen `Comparator`. Die Implementierung basiert typischerweise auf einem Rot-Schwarz-Baum.

5.5.1 Anwendungsbereiche

`TreeSet`s sind nützlich, wenn:

- Eindeutige Elemente in einer sortierten Reihenfolge gespeichert und abgerufen werden müssen.
- Operationen wie das Finden des kleinsten/größten Elements oder das Abrufen von Teilmengen (Ranges) basierend auf der Sortierung benötigt werden.

5.5.2 Vorteile

- Elemente werden automatisch sortiert gehalten.
- Effiziente Operationen ($O(\log n)$) für `add`, `remove` und `contains`.
- Ermöglicht den Zugriff auf das erste/letzte Element sowie auf Teilmengen in $O(\log n)$ oder $O(1)$ (für erste/letzte).

5.5.3 Nachteile

- Langsamer als `HashSet` für die grundlegenden Operationen `add`, `remove` und `contains`, da die Sortierreihenfolge aufrechterhalten werden muss.
- Erfordert, dass Elemente entweder `Comparable` implementieren oder ein `Comparator` bereitgestellt wird. `null`-Elemente sind standardmäßig nicht erlaubt.

5.5.4 Beispiel

Listing 5.5: Beispiel für die Verwendung eines `TreeSet` in Java

```

1 import java.util.TreeSet;
2 import java.util.Set; // Interface verwenden
3 import java.util.SortedSet; // Spezifischeres Interface
4
5 SortedSet<Integer> sortedNumbers = new TreeSet<>();

```

```

6 | sortedNumbers.add(50);
7 | sortedNumbers.add(10);
8 | sortedNumbers.add(90);
9 | sortedNumbers.add(10); // Wird ignoriert, da Duplikat
10
11 | // Iteration erfolgt in sortierter Reihenfolge
12 | for (Integer number : sortedNumbers) {
13 |     System.out.println(number); // Ausgabe: 10, 50, 90
14 | }
15
16 | System.out.println(sortedNumbers.first()); // 10
17 | System.out.println(sortedNumbers.last()); // 90
18 | // System.out.println(sortedNumbers.higher(50)); // gäbe ↵
    |     es nicht für SortedSet, nur für NavigableSet/TreeSet
19 | System.out.println(((TreeSet<Integer>)sortedNumbers). ↵
    |     higher(50)); // 90 (Casting auf TreeSet)

```

5.6 HashMap

Eine HashMap implementiert das Map-Interface und speichert Schlüssel-Wert-Paare. Jedem Schlüssel kann höchstens ein Wert zugeordnet sein. HashMap erlaubt null-Werte und einen einzelnen null-Schlüssel. Die Reihenfolge der Einträge in einer HashMap ist nicht garantiert und kann sich ändern. Die Implementierung basiert auf einer Hashtabelle.

5.6.1 Anwendungsbereiche

HashMaps sind die Standardwahl für Map-Implementierungen, wenn:

- Daten als Schlüssel-Wert-Paare gespeichert werden sollen.
- Ein schneller Zugriff auf Werte über ihre Schlüssel erforderlich ist.
- Die Reihenfolge der Einträge keine Rolle spielt.

5.6.2 Vorteile

- Sehr schnelle durchschnittliche Zeitkomplexität von $O(1)$ (amortisiert) für die Operationen put, get, remove und containsKey.
- Flexibel durch die Erlaubnis eines null-Schlüssels und beliebig vieler null-Werte.

5.6.3 Nachteile

- Keine garantierte Reihenfolge der Schlüssel oder Werte bei der Iteration.
- Ähnlich wie bei HashSet kann die Leistung bei schlechten hashCode()-Implementierungen der Schlüssel oder hoher Füllrate auf $O(n)$ für einzelne Operationen abfallen.

5.6.4 Beispiel

Listing 5.6: Beispiel für die Verwendung einer HashMap in Java

```
1 import java.util.HashMap;
2 import java.util.Map; // Interface verwenden
3
4 Map<String, Integer> ageMap = new HashMap<>();
5 ageMap.put("Alice", 30);
6 ageMap.put("Bob", 25);
7 ageMap.put("Charlie", 35);
8 ageMap.put("Alice", 32); // Wert für "Alice" wird ↵
    aktualisiert
9
10 System.out.println(ageMap.get("Bob")); // 25
11 System.out.println(ageMap.containsKey("David")); // ↵
    false
12 ageMap.put(null, 40); // Null-Schlüssel ist erlaubt
13 ageMap.put("Eve", null); // Null-Wert ist erlaubt
14
15 for (Map.Entry<String, Integer> entry : ageMap.entrySet() ↵
    ()) {
16     System.out.println(entry.getKey() + ":_ " + entry. ↵
        getValue()); // Reihenfolge nicht garantiert
17 }
```

5.6.5 Prüfungshinweis

Dass die HashMap (und hashCode) Effizient funktioniert, muss die hashCode ↵ Methode (und damit die equals Methode) sinnvoll überschrieben werden.

5.7 TreeMap

Eine TreeMap implementiert das SortedMap-Interface (welches Map erweitert) und speichert Schlüssel-Wert-Paare in sortierter Reihenfolge der Schlüssel. Die Sortierung der Schlüssel erfolgt entweder natürlich (wenn die Schlüssel Comparable implementieren) oder durch einen beim Erstellen der TreeMap übergebenen Comparator. TreeMap erlaubt keine null-Schlüssel (im Gegensatz zu HashMap), aber null-Werte sind erlaubt. Die Implementierung basiert typischerweise auf einem Rot-Schwarz-Baum.

5.7.1 Anwendungsbereiche

TreeMaps werden verwendet, wenn:

- Schlüssel-Wert-Paare gespeichert werden müssen und eine Iteration über die Schlüssel in sortierter Reihenfolge erforderlich ist.
- Operationen benötigt werden, die von der Sortierung der Schlüssel abhängen, wie das Finden des Eintrags mit dem kleinsten/größten Schlüssel oder das Abrufen von Teilmengen basierend auf Schlüsselbereichen.

5.7.2 Vorteile

- Schlüssel werden automatisch sortiert gehalten.
- Effiziente Operationen ($O(\log n)$) für put, get, remove und containsKey.
- Ermöglicht den Zugriff auf den ersten/letzten Eintrag sowie auf Teil-Maps (Ranges) in $O(\log n)$ oder $O(1)$.

5.7.3 Nachteile

- Langsamer als HashMap für die grundlegenden Operationen, da die Sortierreihenfolge der Schlüssel aufrechterhalten werden muss.
- Erfordert, dass Schlüssel entweder Comparable implementieren oder ein Comparator bereitgestellt wird.
- Erlaubt keine null-Schlüssel (wirft NullPointerException).

5.7.4 Beispiel

Listing 5.7: Beispiel für die Verwendung einer TreeMap in Java

```
1 import java.util.TreeMap;
2 import java.util.Map; // Interface verwenden
3 import java.util.SortedMap; // Spezifischeres Interface
4
5 SortedMap<Integer, String> sortedUserMap = new TreeMap <-
    <>();
6 // Alternativ: Map<Integer, String> sortedUserMap = new <-
    TreeMap<>();
7
8 sortedUserMap.put(102, "Alice");
9 sortedUserMap.put(100, "Bob");
10 sortedUserMap.put(105, "Charlie");
11 // sortedUserMap.put(null, "David"); // Würde <-
    NullPointerException werfen
12
13 // Iteration erfolgt in sortierter Reihenfolge der Schlü <-
    ssel
14 for (Map.Entry<Integer, String> entry : sortedUserMap. <-
    entrySet()) {
15     System.out.println(entry.getKey() + ":\u0020" + entry. <-
        getValue()); // 100: Bob, 102: Alice, 105: <-
        Charlie
16 }
17
18 System.out.println(sortedUserMap.firstKey()); // 100
19 // Für subMap etc. ist SortedMap ausreichend
20 SortedMap<Integer, String> subMap = sortedUserMap.subMap <-
    (100, 103); // Einträge für 100 (inkl.), 102 (inkl.), <-
    103 (exkl.)
21 System.out.println("SubMap:\u0020" + subMap);
```


5.8 Über Collections Iterieren (Prüfungshinweis)

Über Collections, abgesehen von der `ArrayList`, sollte *immer* mittels Iterator iteriert werden. Andere lösungen mit einer Zähl Variable und mehreren `.get` aufrufen funktionieren zwar, sind aber in der Praxis sehr Ineffizient und führen zu Punktabzug.

6. Hüllenklassen

6.1 Definition und Notwendigkeit

Hüllenklassen umhüllen primitive Datentypen. Dies ist beispielsweise für die Collection Klassen notwendig, da diese nur Objekte als Parameter aufnehmen. Hüllklassen haben keine andere Aufgabe als einen primitiven Datentyp zu umhüllen.

6.2 Übersicht der Hüllenklassen

Die folgende Tabelle zeigt die primitiven Datentypen und ihre entsprechenden Hüllenklassen:

Primitiver Datentyp	Hüllklasse
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

6.3 Erstellung von Hüllenklassen

Hüllenklassen werden wie jedes andere Objekt mittels des new-Operators instanziiert:

Listing 6.1: Erstellung von Hüllenklassen-Objekten

```
1 Integer i = new Integer(10);  
2 Boolean b = new Boolean(true);
```

6.4 Verwendung vor JDK 1.5

Bis zum Java Development Kit (JDK) Version 1.4 war eine direkte Kombination von Hüllenklassen-Objekten mit primitiven Datentypen in arithmetischen oder logischen Ausdrücken nicht möglich.

6.4.1 Ungültige Ausdrücke (Theoretisch)

Die folgenden Operationen wären ohne explizites Auspacken ungültig gewesen:

Listing 6.2: Theoretisch ungültige Ausdrücke vor JDK 1.5

```
1 Integer iObj = new Integer(10);
2 Boolean bObj = new Boolean(true);
3
4 // iObj + 20; // Compilerfehler: Operator + nicht ↔
   // anwendbar auf Integer, int
5 // bObj && true; // Compilerfehler: Operator && nicht ↔
   // anwendbar auf Boolean, boolean
```

6.4.2 Explizites Entpacken (Unboxing)

Um den Wert eines Hüllenklassen-Objekts verwenden zu können, musste dieser explizit mittels einer entsprechenden Methode (z.B. `intValue()`) entpackt werden:

Listing 6.3: Explizites Entpacken des Wertes

```
1 Integer iObj = new Integer(10);
2 int j = iObj.intValue(); // j erhält den Wert 10
3
4 Boolean bObj = new Boolean(true);
5 boolean bVal = bObj.booleanValue(); // bVal erhält den ↔
   // Wert true
```

6.5 Autoboxing und Autounboxing (seit JDK 1.5)

Seit dem JDK 1.5 führt der Java-Compiler das notwendige Umwandeln zwischen primitiven Typen und ihren Hüllenklassen automatisch durch. Dies wird als Autoboxing (primitiv zu Hüllklasse) und Autounboxing (Hüllklasse zu primitiv) bezeichnet. Der Compiler fügt die notwendigen Aufrufe (z.B. `intValue()`) automatisch ein, wodurch der Code lesbarer wird:

Listing 6.4: Automatische Umwandlung durch Autoboxing/Autounboxing

```
1 Integer i = new Integer(10); // Explizite Erstellung ( ↔
   // Boxing auch möglich: Integer i = 10;)
2 Boolean b = new Boolean(true); // Explizite Erstellung ( ↔
   // Boxing auch möglich: Boolean b = true;)
3
4 int k = i + 20; // Autounboxing: i wird zu int entpackt
5 if (b && true) { // Autounboxing: b wird zu boolean ↔
   // entpackt
6 // ...
7 }
```

6.6 Umwandlung von Strings in Hüllenklassenwerte

Hüllenklassen bieten statische Methoden (z.B. `parseInt()`), um Strings, die Zahlen oder Wahrheitswerte repräsentieren, in den entsprechenden primitiven Datentyp oder direkt in ein Hüllenklassen-Objekt umzuwandeln.

Listing 6.5: Parsen von Strings zu primitiven Typen/Hüllenklassen

```
1 int i = Integer.parseInt("1337");
2 double d = Double.parseDouble("-1.25E-13");
3 boolean b = Boolean.parseBoolean("true");
4
5 // Erzeugt direkt Hüllenklassen-Objekte (oft weniger ←
   // gebräuchlich als das Parsen zu primitiven Typen)
6 // Integer iObj = Integer.valueOf("1337");
7 // Double dObj = Double.valueOf("-1.25E-13");
8 // Boolean bObj = Boolean.valueOf("true");
```

7. Exceptions (Ausnahmen)

Eine **Exception** (Ausnahme) ist ein abnormales oder unerwartetes Ereignis, das während der Ausführung eines Programms auftritt und den normalen Programmfluss unterbricht. Exceptions signalisieren Fehlerzustände, die zur Laufzeit auftreten. Sie können durch eine Vielzahl von Situationen ausgelöst werden, beispielsweise durch:

- **Knappe Ressourcen:** Zu wenig Arbeitsspeicher (z.B. `OutOfMemoryError`) oder Festplattenspeicher.
- **Ungültige Operationen:**
 - Unzulässiger Zugriff auf ein Array (z.B. Zugriff auf einen Index außerhalb der Array-Grenzen, was zu einer `ArrayIndexOutOfBoundsException` führen kann).
 - Versuch, eine Methode auf einer `null`-Referenz aufzurufen (was eine `NullPointerException` auslöst).
 - Ungültige Typkonvertierung zwischen inkompatiblen Typen (kann eine `ClassCastException` verursachen).
- **Fehlerhafte Eingaben:** Verarbeitung von ungültigen oder unerwarteten Daten.
- **Externe Fehler:** Probleme mit Netzwerkverbindungen, Dateizugriffen (z.B. Datei nicht gefunden - `FileNotFoundException`).
- **Verletzung einer Zusicherung:** Das Scheitern einer `assert`-Anweisung (führt zu einem `AssertionError`), was typischerweise auf einen Programmierfehler hindeutet.

7.1 Folgen einer Exception

Wenn eine Exception während der Programmausführung auftritt und nicht behandelt wird, hat dies meist gravierende Folgen:

1. **Programmabbruch:** Die Methode, in der die Exception aufgetreten ist, wird sofort beendet. Wenn die Exception auch von keiner der aufrufenden Methoden in der Aufrufkette (Call Stack (Aufrufstapel / Aufrufkamin)) behandelt wird, terminiert das gesamte Programm abrupt. Man spricht oft davon, dass das Programm „abstürzt“.

2. **Fehlermeldung:** In der Regel wird eine Fehlermeldung ausgegeben, die Informationen über den Typ der Exception und die Stelle im Code enthält, an der sie aufgetreten ist (Stack Trace (Aufrufliste / Stapelrückverfolgung)).

Um einen solchen Absturz zu verhindern und stattdessen kontrolliert auf Fehler reagieren zu können, kommt die Ausnahmebehandlung (Exception Handling) ins Spiel.

7.2 Ausnahmebehandlung mit Try-Catch-Finally-Blöcken

Zur Behandlung von Exceptions werden spezielle Code-Blöcke verwendet: `try`, `catch` und optional `finally`.

7.2.1 Der `try`-Block

Der `try`-Block umschließt den Code-Abschnitt, in dem eine Exception auftreten könnte. Dieser Code wird als "kritischer Abschnitt" betrachtet.

Listing 7.1: Beispiel für einen `try`-Block

```
1 try {  
2     // Kritischer Code, der möglicherweise eine ←  
3     Exception auslöst  
4     // z.B. int result = 10 / 0; // Würde eine ←  
5     ArithmeticException auslösen  
6     // oder  
7     // String text = null;  
8     // System.out.println(text.length()); // Würde eine ←  
9     NullPointerException auslösen  
10 } catch (SpecificExceptionType1 e1) {  
11     // Behandlung für SpecificExceptionType1  
12 }  
13 // ... weitere catch-Blöcke oder finally-Block
```

7.2.2 Der `catch`-Block

Wenn innerhalb des `try`-Blocks eine Exception auftritt, wird die normale Ausführung des `try`-Blocks sofort abgebrochen. Das Laufzeitsystem sucht dann nach einem passenden `catch`-Block, der diese spezifische Exception oder eine ihrer Oberklassen behandeln kann.

- Ein `try`-Block kann einen oder mehrere `catch`-Blöcke haben.
- Jeder `catch`-Block ist für einen bestimmten Typ von Exception zuständig.
- Die `catch`-Blöcke werden in der Reihenfolge geprüft, in der sie definiert sind. Der erste `catch`-Block, dessen Exception-Typ zur aufgetretenen Exception passt, wird ausgeführt.
- Nachdem ein `catch`-Block ausgeführt wurde, wird die Ausführung nach dem gesamten `try-catch`-Konstrukt fortgesetzt (es sei denn,

der catch-Block selbst löst eine neue Exception aus oder beendet das Programm).

- Wird keine Exception im try-Block geworfen, werden alle catch-Blöcke übersprungen.

Beispiel für **catch**-Blöcke:

Listing 7.2: Beispiel für catch-Blöcke

```
1 // Angenommen, wir lesen Daten aus einer Datei und ↵  
  verarbeiten Zahlen  
2 try {  
3     // Code, der potenziell eine IOException (z.B. Datei ↵  
        nicht gefunden)  
4     // oder eine NumberFormatException (z.B. Text kann ↵  
        nicht in Zahl umgewandelt werden) auslösen könnte ↵  
        .  
5     String daten = dateiLesen("eingabe.txt"); // Methode ↵  
        könnte IOException werfen  
6     int zahl = Integer.parseInt(daten); // Könnte ↵  
        NumberFormatException werfen  
7     System.out.println("Verarbeitete_Zahl:_ " + zahl);  
8  
9 } catch (java.io.IOException e) {  
10    // Spezifische Behandlung, wenn ein Fehler beim ↵  
        Dateizugriff auftritt  
11    System.err.println("Fehler_beim_Zugriff_auf_die_ ↵  
        Datei:_ " + e.getMessage());  
12    // Hier könnte man z.B. einen Standardwert verwenden ↵  
        oder den Benutzer informieren.  
13  
14 } catch (NumberFormatException e) {  
15    // Spezifische Behandlung, wenn die Daten nicht in ↵  
        eine Zahl konvertiert werden können  
16    System.err.println("Ungültiges_Zahlenformat_in_den_ ↵  
        Daten:_ " + e.getMessage());  
17    // Hier könnte man den Benutzer um eine korrigierte ↵  
        Eingabe bitten.  
18  
19 } catch (Exception e) { // Ein allgemeinerer Exception- ↵  
    Handler  
20    // Dieser Block fängt alle anderen Exceptions, die ↵  
        von den vorherigen catch-Blöcken nicht abgefangen ↵  
        wurden.  
21    // Es ist oft eine gute Praxis, spezifischere ↵  
        Exceptions zuerst zu fangen.  
22    System.err.println("Ein_unerwarteter_Fehler_ist_ ↵  
        aufgetreten:_ " + e.getMessage());  
23    e.printStackTrace(); // Gibt den \gls{stacktrace} ↵  
        aus, um bei der Fehlersuche zu helfen.  
24 }
```

Im obigen Beispiel wird `e` (oder `e1`, `e2` etc.) als Objekt der jeweiligen Exception-Klasse deklariert. Dieses Objekt enthält Informationen über den

Fehler, wie z.B. eine Fehlermeldung (`e.getMessage()`) oder den Stack Trace (Aufrufliste / Stapelrückverfolgung).

7.2.3 Der **finally**-Block (optional)

Ein `finally`-Block kann an einen `try`-Block (nach allen `catch`-Blöcken) angehängt werden. Der Code innerhalb eines `finally`-Blocks wird **immer** ausgeführt, unabhängig davon, ob:

- eine Exception im `try`-Block aufgetreten ist oder nicht.
- eine aufgetretene Exception von einem `catch`-Block abgefangen wurde oder nicht.
- ein `catch`-Block oder der `try`-Block durch eine `return`-Anweisung verlassen wird.

Der `finally`-Block wird typischerweise für Aufräumarbeiten verwendet, wie z.B. das Schließen von Dateien, Netzwerkverbindungen oder Datenbankverbindungen, um sicherzustellen, dass Ressourcen freigegeben werden, egal was passiert.

Listing 7.3: Beispiel für einen `finally`-Block

```
1 java.io.FileReader reader = null;
2 try {
3     reader = new java.io.FileReader("meineDatei.txt");
4     // ... Code zum Lesen der Datei ...
5     // Kann eine IOException auslösen
6 } catch (java.io.IOException e) {
7     System.err.println("Fehler_beim_Lesen_der_Datei:_" +
8         e.getMessage());
9 } finally {
10     if (reader != null) {
11         try {
12             reader.close(); // Wichtig: Ressourcen
13                             // freigeben!
14         } catch (java.io.IOException e) {
15             System.err.println("Fehler_beim_Schliessen_
16                             // des_Readers:_" + e.getMessage());
17         }
18     }
19     System.out.println("Finally-Block_wurde_ausgefuehrt." +
20         "\n");
21 }
```

7.3 Unbehandelte Exceptions

Exceptions, welche nicht explizit durch einen `catch`-Block in der aktuellen Methode behandelt werden, werden automatisch an die aufrufende Methode weitergegeben (propagiert). Dieser Vorgang setzt sich die Aufrufkette (Call Stack (Aufrufstapel / Aufrufkamin)) hinauf fort.

Wenn eine Exception bis zur `main`-Methode (dem Einstiegspunkt des Programms) propagiert und auch dort nicht behandelt wird, führt dies zum Ab-

bruch des gesamten Programms. Dabei wird üblicherweise der Stack Trace (Aufrufliste / Stapelrückverfolgung) der Exception auf der Konsole ausgegeben, um die Fehlerquelle zu identifizieren.

8. Schnittstellen (Prüfungsrelevant)

Schnittstellen geben ein Versprechen, dass Klassen bestimmte Fähigkeiten besitzen, ohne vorzugeben, wie sich eine Klasse zu verhalten hat. Sie definieren, welche Methoden eine Klasse implementieren muss.

Listing 8.1: Beispiel einer Schnittstelle

```
1
2 public interface Salable {
3     boolean matches(String searchPattern);
4     String asText();
5     float getPrice();
6 }
```

8.1 Eigenschaften und Hinweise

- Schnittstellen können **nicht** `protected` oder `private` sein.
- Es gibt keine oberste Schnittstelle, analog zur Klasse `Object`.
- In Schnittstellen können `public static final`-Variablen (Konstanten) deklariert werden.
- Schnittstellen können `static` Methoden besitzen, die keinen Bezug zu Objekten haben.
- Wenn eine Klasse eine Schnittstelle implementiert (`implements`), muss sie entweder `abstract` sein oder alle nicht-statischen Methoden der Schnittstelle definieren.

8.2 Schnittstellen als Typ

Schnittstellen können als Typdeklaration und als Ergebnistyp von Methoden verwendet werden. Dies ermöglicht die Angabe eines Objekts jeder Klasse, die die Schnittstelle implementiert.

Listing 8.2: Schnittstelle als Typ

```
1 Salable item;
2
3 public void add(Salable item) {
```

```

4 |     ...
5 | }

```

8.3 Funktionale Schnittstellen

Eine Schnittstelle ist funktional, wenn sie genau eine nicht-statische Methode definiert. Sie kann optional mit der Annotation `@FunctionalInterface` versehen werden.

8.4 Lambda-Ausdrücke (sehr sehr Prüfungsrelevant)

Lambda-Ausdrücke sind namenlose Funktionen, die über die Syntax `(lambdaParameter) -> Anweisung` definiert werden.

8.4.1 Vereinfachung von Lambda-Ausdrücken

Lambda-Ausdrücke können vereinfacht werden:

- Der Typ der Parameter kann weggelassen werden, da er aus dem Kontext hervorgeht.
- Geschweifte Klammern können weggelassen werden, wenn der Ausdruck nur eine Anweisung enthält.
- Runde Klammern können weggelassen werden, wenn der Ausdruck nur einen Parameter hat.

Listing 8.3: Vereinfachung von Lambda ausdrücken

```

1 (String s) -> { return s.length() >= 4; } // Ursprü ←
   nglich
2
3 (s) -> { return s.length() >= 4; } // Parameter ←
   Typen weglassen
4 (s) -> s.length() >= 4; // Geschweifte ←
   Klammern weglassen
5 s -> s.length() >= 4; // Runde ←
   Klammern weglassen

```

Wenn in einem Lambda-Ausdruck nur eine Methode aufgerufen wird, kann eine Methodenreferenz verwendet werden:

Listing 8.4: Methodenreferenz

```

1 s -> s.isEmpty(); // Lambda Ausdruck
2 String::isEmpty; // Methodenreferenz

```

8.4.2 Hinweise zu Lambda-Ausdrücken

Lambda-Ausdrücke dürfen lokale Variablen nicht verändern, können aber deren Werte auslesen. Objekte können über Methodenaufrufe bearbeitet werden.

9. Streams

Ein Stream ist eine Folge von Elementen, auf die sequentielle und parallele Operationen angewendet werden können.

Streams unterstützen eine Vielzahl an Operationen wie Filtern, Transformieren, Aggregieren und vieles mehr. Zudem lassen sich Streams parallel verarbeiten, wodurch die Laufzeit im Vergleich zu normalen Schleifen auf Mehrkern-CPUs deutlich schneller ist.

9.1 Beispiel

Listing 9.1: Beispiel für Streams

```
1 Stream.of("08/15", "4771", "501", "s04", "1250", "333", ←  
   "475")  
2     .filter(s -> s.matches("[0-9]+")) // nur Strings ←  
   welche ausschließlich Ziffern enthalten  
3     .map(s -> Integer.parseInt(s)) // Strings als ←  
   Integer parsen  
4     .filter(n -> n >= 400) // nur Zahlen größer gleich ←  
   400  
5     .sorted() // Zahlen sortieren  
6     .forEach(System.out::println); // Zahlen ausgeben
```

Die Ausgabe des Streams ist dann '475, 501, 1250, 4771'.

9.2 Arten von Streams

Es gibt vier verschiedene Arten von Streams:

- Stream<T>
- IntStream
- DoubleStream
- LongStream

Der Stream `Stream<Integer>` ist **nicht** äquivalent zu dem `IntStream` ←
. Der `IntStream` besitzt einige Methoden wie beispielsweise `sum`, welche der `Stream<Integer>` nicht besitzen kann.

9.2.1 IntStream Beispiel

Listing 9.2: Beispiel eines IntStreams

```
1 IntStream.rangeClosed(111, 999)
2     .filter(n -> n % (n / 100 + n / 10 % 10 + n % 10) == 0)
3     .forEach(System.out::println);
```

Hier wird ein `IntStream` mit dem geschlossenen Intervall (111, 999) erstellt, bei welchem dann alle Zahlen gefiltert werden, die durch die Summe ihrer Ziffern teilbar sind. Diese werden anschließend in der Konsole ausgegeben.

9.3 Hinweis

Ein Stream ist **keine** Datenstruktur. Sie verwalten keine Daten. Wenn ein Stream basierend auf einer Collection erstellt wird und diese Collection anschließend bearbeitet wird, dann verändert sich der Stream automatisch mit.

Listing 9.3: Stream über Collection

```
1 ArrayList<Integer> list = new ArrayList<>();
2 list.add(10);
3 list.add(20);
4 list.add(30);
5 Stream s = list.stream();
6 list.remove(0);
7 s.forEach(System.out::println);
```

Es wird ‘20, 30‘ in der Konsole ausgegeben, da der Stream die Änderungen in der zugrunde liegenden Collection widerspiegelt.

9.4 Stream erzeugen (Häufige Prüfungsaufgabe)

Ein Stream kann über mehrere Wege erzeugt werden:

- Über die statische Methode in der Schnittstelle `Stream<T>: Stream <T> of(T... values)`
- Über die Instanzmethode der Schnittstelle `Collection<E>: Stream <E> stream() → ArrayList<String> list.stream()`
- Über die statische Methode in der Klasse `Arrays: Arrays.stream <T> (new String[])`
- Über die Instanzmethode der Klasse `BufferedReader: BufferedReader <T> br.lines()`

9.4.1 Iterate

Es kann eine Funktion verwendet werden, um einen unendlichen Stream zu erstellen:

Listing 9.4: Unendlicher Stream mit Iterate

```
1 Stream<Integer> s = Stream.iterate(2, n -> n + 2). ←
    takeWhile(n -> n > 0);
```

Somit wurde ein Stream erstellt, welcher alle positiven geraden Zahlen beinhaltet. Das `takeWhile(n -> n > 0)` sorgt dafür, dass nur positive Zahlen ausgegeben werden und der Stream terminiert, sobald `n` durch einen Integer-Overflow negativ wird.

im IntStream

Listing 9.5: Unendlicher Stream mit Iterate im IntStream

```
1 IntStream is = IntStream.iterate(2, n -> n + 2). ←
    takeWhile(n -> n > 0);
```

9.4.2 im LongStream

Listing 9.6: Unendlicher Stream mit Iterate im LongStream

```
1 LongStream ls = LongStream.iterate(2, n -> n + 2). ←
    takeWhile(n -> n > 0);
```

9.4.3 Generate

Ein Stream kann mit einer `generate`-Methode gekoppelt werden, welche aufgerufen wird, um ein neues Element anzufragen:

Listing 9.7: Unendlicher Stream mit Generate

```
1 // Beispiel für eine Methode in einer Klasse, die ein ←
    Integer zurückgibt
2 private int n = 0; // Das Feld muss im Kontext der ←
    Klasse definiert sein
3
4 public Integer nextInt() {
5     return n++; // n zurückgeben und danach erhöhen
6 }
7
8 // Aufruf im Kontext der Klasse (z.B. in einer anderen ←
    Methode)
9 Stream<Integer> s = Stream.generate(this::nextInt). ←
    takeWhile(n -> n >= 0);
10 IntStream is = IntStream.generate(this::nextInt). ←
    takeWhile(n -> n >= 0);
11 LongStream ls = LongStream.generate(this::nextInt). ←
    takeWhile(n -> n >= 0);
```

Es wird ein Stream erzeugt, welcher über alle positiven `int`-Werte läuft. Das `takeWhile(n -> n > 0)` sorgt dafür, dass nur positive Zahlen im Stream landen und der Stream bei einem möglichen Integer-Overflow terminiert.

9.4.4 IntStream.range

Ein `IntStream` und `LongStream` kann auch über ein Intervall erstellt werden. Hier gibt es die statischen Methoden `range(int startInclusive, int endExclusive)` und `rangeClosed(int startInclusive, int endInclusive)`. Die erstellen ein Intervall von `start` bis `end`. Der Unterschied zwischen den beiden Methoden ist, dass bei der `range` Methode das Ende nicht mit eingeschlossen ist und bei der `rangeClosed` das ende mit eingeschlossen ist.

9.5 Operationen auf Streams

Auf streams können eine vielzahl an Operationen durchgeführt werden. Die wichtigsten sind:

- `limit(int limit)` Geht mit maximal so vielen Elementen über den Stream. Hat der Stream weniger Elemente als das limit dann hört er auf, bevor das limit erreicht wurde.
- `filter(Predicate<? super T> predicate)` Filtert den Stream nach bestimmten kriterien. Beispiel: `Stream.filter(i -> i > 3)`
- `map(Function<? super T, extends R> mapper)` Mappt die Elemente des Streams. Beispiel: `Stream.map(s -> Integer.parseInt(s))`
- `mapToInt` Ist äquivalent zu `Stream.map(s -> Integer.parseInt(s))`, nur dass ein `IntStream` und kein `Stream<Integer>` zurückgegeben wird. Es gibt äquivalente Methoden `mapToLong` und `mapToDouble`
- `min(Comparator<? super T> comparator)` und `max(Comparator<? super T> comparator)` Gibt den Maximalen bzw Minimalen Wert zugrunde des Comparators. Comparator funktioniert wie das `compareTo`
- `findFirst()` Gibt das Erste Element das gefunden aus. Hier wird ein Optional zurückgegeben.
- `findAny()` Gibt irgend ein Element des Streams als Optional zurück.
- `anyMatches(Predicate<? super T> predicate)` Gibt als Boolean aus, ob irgendein Element des Streams eine bedingung erfüllt. Beispiel: `Stream.anyMatches(s -> s.length() < 3)`
- `allMatches(Predicate<? super T> predicate)` äquivalent zu `anyMatches`, nur dass geprüft wird, ob Alle elemente des Streams diese bedingung erfüllen.
- `noneMatch(Predicate<? super T> predicate)` äquivalent zu `anyMatches`, nur dass geprüft wird, ob kein Element der Liste diese bedingung erfüllt.
- `sorted(<Comparator<? super T> comparator)` Sortiert den Stream entweder aufgrund des `compareTo` (wenn kein Parameter übergeben

wird) oder anhand eines Comperator. Beispiel: `Stream.sort((s1 ↔
, s2) -> s1.toLowerCase().compareTo(s2.toLowerCase()))`

10. Ein- und Ausgabe

10.1 InputStream

Der `InputStream` ist die Oberklasse und Abstraktion aller byteorientierten Eingabeströme. Es gibt nur zwei Methoden zum Lesen von Bytes:

- `int read()` liest *ein* Byte
- `int read(byte[])` liest mehrere Bytes, Ergebniswert hat eine andere Bedeutung als der von `read()`

Der Rückgabewert der `read()` Methode gibt den gelesenen Byte zurück. Dieser kann einen Wert zwischen 0 und 255 (dem gelesenen Byte) sein. Ein Rückgabewert von -1 signalisiert, dass das Ende des Streams erreicht wurde und keine weiteren Bytes verfügbar sind. Es wird also keine Fehlermeldung geworfen, wenn das Ende der Datei erreicht wurde. Wenn bei dem Lesen allerdings ein Fehler auftritt, wird eine `IOException` geworfen.

Listing 10.1: Beispiel für InputStream

```
1  InputStream is = new ByteArrayInputStream(new byte[]{1, 2, 3, 4, 5});
2
3  int b = is.read();
4  ArrayList<Integer> content = new ArrayList<>();
5
6  while (b != -1) {
7      content.add(b);
8  }
```

Das einzelne Lesen von Bytes aus einem `InputStream` ist sehr ineffizient. Dateien können sehr groß werden, schnell auch mehrere Gigabyte groß. Dies entspricht Millionen von `read()` aufrufen und damit Millionen von I/O-Zugriffen. Aufgrund dessen ist die einfache `read()` Methode in `InputStreams` oft eine schlechte Lösung. Abhilfe schafft hier die Klasse `BufferedInputStream`.

10.1.1 BufferedInputStream

Der `BufferedInputStream` kapselt einen `InputStream` und liest intern aus diesen Blockweise über die Methode `read(byte[])`. Hier ist die Methode `read()` effizient und kann gut genutzt werden.

10.1.2 ByteArrayInputStream

Dies ist beispielsweise essenziell für das Schreiben von Testklassen, da diese niemals direkt von externen Dateien abhängen sollten, um in sich geschlossen und reproduzierbar zu sein.

10.2 Reader

Ähnlich wie der `InputStream` bietet der `Reader` zwei Methoden zum Lesen von Zeichen:

- `int read()` liest *ein* Zeichen
- `int read(char[])` liest mehrere Zeichen, Ergebniswert hat eine andere Bedeutung als der von `read()`

Auch hier ist die Methode `read()` für relativ große Datenquellen sehr ineffizient.

10.2.1 BufferedReader

Der `BufferedReader` kapselt einen `Reader` ein und liest intern auf diesem blockweise. Die `read()`-Methode ist in dieser Klasse durch das interne Puffern auch für einzelne Zeichenlesevorgänge effizient.

Der `BufferedReader` enthält außerdem die Methode `String readLine()` um eine ganze Zeile zu lesen. Dies kann auch der gesamte Inhalt der Datenquelle sein. Außerdem enthält er die Methode `Stream<String> lines()` um einen Stream aller Zeilen zu erstellen.

10.2.2 StringReader

Die Klasse `StringReader` eignet sich sehr gut um Methoden mit `Reader`-Parametern zu testen.

10.3 Codierung

Es gibt viel mehr Zeichen als Bytes. Deshalb müssen einige Zeichen durch mehrere Bytes repräsentiert werden. Eine Zeichencodierung beschreibt die Art der Codierung von Zeichen durch Bytes. Die am häufigsten verwendete Codierung ist UTF-8. Die meisten westeuropäischen Zeichen werden hier durch ein Byte dargestellt. Alle anderen Zeichen durch 2 bis 4 Byte. In UTF-16 werden alle Zeichen durch mindestens zwei Byte dargestellt.

Um die Codierung eines `InputStreams` anzugeben, hat sein Konstruktor einen zweiten Parameter. Über den Aufzählungstypen `StandardCharsets` kann hier dann die erwünschte Codierung angegeben werden.

Listing 10.2: Beispiel für Codierten `InputStream`

```
1 ByteArrayInputStream byteStream = new  
    ByteArrayInputStream("Test_mit_ÄÖÜ".getBytes(  
        StandardCharsets.UTF_8));
```

```

2 Reader reader = new InputStreamReader(byteStream, ↵
    StandardCharsets.UTF_8);
3 StringBuilder result = new StringBuilder();
4 int character;
5
6 try {
7     while ((character = reader.read()) != -1) {
8         result.append((char) character);
9     }
10 } finally {
11     reader.close();
12 }

```

Verfügbare StandardCharsets sind:

- **ISO_8859_1**
- **US_ASCII**
- **UTF_16**
- **UTF_16BE**
- **UTF_16LE**
- **UTF_8**

Die Codierung eines Readers kann nicht geändert werden, da dieser bereits Zeichen liefert und eben keine Bytes. Allerdings kann über die Instanzmethode `byte[] getBytes()` der String Klasse die Codierung angepasst werden:

Listing 10.3: Beispiel für String Codierung ändern

```

1 String string = "examplestring";
2 byte[] bytes = string.getBytes();
3
4 String asciiEncodedString = new String(bytes, ↵
    StandardCharsets.US_ASCII);
5
6 System.out.println(asciiEncodedString);
7
8 String utf8EncodedString = new String(bytes, ↵
    StandardCharsets.UTF_8);
9
10 System.out.println(utf8EncodedString);
11
12 String utf16EncodedString = new String(bytes, ↵
    StandardCharsets.UTF_16);
13
14 System.out.println(utf16EncodedString);

```

Wenn bytes in einer Tabelle nicht gefunden werden, werden diese über ein ? repräsentiert. Wird eine falsche Codierung angewandt, so werden die Zeichen anders interpretiert. Die Methode `getBytes()` übersetzt den String standardmäßig in die Standardcodierung der JVM, welche in Deutschland idr. UTF-8 ist. Diese Codierung kann allerdings über den Parameter explizit angegeben werden.

10.4 OutputStream

Der OutputStream ist die Oberklasse aller Byteorientierten Ausgabeströmen. Er besitzt zwei Methoden:

- `write(int)`
- `write(byte[])`

Die Methode `write(int)` ist bei externen Datensinken potentiell Ineffizient. Aufgrunddessen gibt es hier die Klasse `BufferedOutputStream`.

Listing 10.4: Beispiel für OutputStream

```
1 OutputStream bs = new ByteArrayOutputStream();
2
3 byte[] bytes = new byte[]{1, 2, 3, 4, 5}
4
5 for (byte b : bytes) {
6     bs.write(b);
7 }
```

10.4.1 BufferedOutputStream

Der `BufferedOutputStream` kapselt einen `OutputStream` und schreibt dort blockweise hinein. Die `write(int)` Methode dieser Klasse ist auch für externe Datensinken effizient.

10.4.2 ByteArrayOutputStream

der `ByteArrayOutputStream` eignet sich zum Testen von Methoden mit `OutputStream`-Parametern.

11. Aufzählungstypen

12. Collection-Klassen (Sehr Prüfungsrelevant)

Arrays haben eine feste Größe, die sich während der Laufzeit nicht ändern kann. Dies stellt häufig eine Einschränkung dar. Sogenannte Collection-Klassen (oder Kollektionen) bieten hier flexiblere Alternativen. Es gibt verschiedene Arten von Collection-Klassen, die sich grob in Listen (`List`), Mengen (`Set`) und Abbildungen (`Map`) einteilen lassen. Jede Art und ihre spezifischen Implementierungen haben eigene Vor- und Nachteile, die sie für unterschiedliche Anwendungsfälle geeignet machen.

12.1 Arrays

Arrays sind streng genommen keine Collection-Klassen im Sinne des Java Collections Frameworks. Dennoch ist es sinnvoll, sie hier zu betrachten, da sie einen grundlegenden Mechanismus zur Gruppierung von Elementen darstellen und als Vergleichsbasis für die eigentlichen Collection-Klassen dienen.

12.1.1 Anwendungsbereiche

Arrays eignen sich, wenn eine feste Anzahl von Elementen desselben Typs geordnet gespeichert werden soll und sich diese Anzahl zur Laufzeit nicht ändert.

12.1.2 Vorteile

Der Zugriff auf Elemente über ihren Index ist bei Arrays sehr schnell und effizient (konstante Zeitkomplexität, $O(1)$).

12.1.3 Nachteile

Die Größe eines Arrays ist nach seiner Initialisierung unveränderlich. Eine nachträgliche Größenänderung erfordert die Erstellung eines neuen Arrays und das Kopieren der Elemente.

12.1.4 Beispiel

Listing 12.1: Beispiel für die Verwendung eines Arrays in Java

```
1 int[] numbers = new int[5]; // Array der Größe 5 ←  
   erstellen
```

```

2 numbers[3] = 42; // Dem Element am Index 3 den Wert 42 ↔
   zuweisen
3 System.out.println(numbers[3]); // Wert am Index 3 ↔
   ausgeben (Ausgabe: 42)
4 // numbers[5] = 7; // Dies würde eine ↔
   ArrayIndexOutOfBoundsException auslösen

```

12.2 ArrayList

ArrayLists sind dynamische Arrays, d.h., sie können ihre Größe während der Laufzeit anpassen. Sie implementieren das List-Interface.

12.2.1 Anwendungsbereiche

ArrayLists sind dann sinnvoll, wenn Elemente geordnet gespeichert werden sollen und sich die Anzahl der Elemente dynamisch ändern kann. Sie bieten einen guten Kompromiss zwischen flexiblem Größenmanagement und schnellem Indexzugriff.

12.2.2 Vorteile

- Die Größe der Liste kann zur Laufzeit dynamisch wachsen oder schrumpfen.
- Der Zugriff auf Elemente über ihren Index ist in der Regel schnell (amortisiert $O(1)$), ähnlich wie bei Arrays.

12.2.3 Nachteile

- Das Einfügen oder Löschen von Elementen am Anfang oder in der Mitte der Liste kann langsam sein ($O(n)$), da nachfolgende Elemente verschoben werden müssen.
- Beim Überschreiten der internen Kapazität muss das zugrundeliegende Array vergrößert und die Elemente kopiert werden. Obwohl das Hinzufügen am Ende amortisiert $O(1)$ ist, kann diese Operation vereinzelt länger dauern.

12.2.4 Beispiel

Listing 12.2: Beispiel für die Verwendung einer ArrayList in Java

```

1 import java.util.ArrayList; // Import notwendig
2 import java.util.List; // Interface verwenden ist gute ↔
   Praxis
3
4 List<Integer> list = new ArrayList<>(); // Neue, leere ↔
   ArrayList erstellen
5 list.add(10); // Element 10 am Ende hinzufügen
6 list.add(20); // Element 20 am Ende hinzufügen
7 list.add(0, 5); // Element 5 am Index 0 einfügen ( ↔
   verschiebt andere Elemente)
8

```

```

9 | System.out.println(list.get(1)); // Element am Index 1  ↔
   |     ausgeben (Ausgabe: 10)
10 | list.remove(0); // Element am Index 0 entfernen

```

12.3 LinkedList

Eine `LinkedList` speichert ihre Elemente in einer doppelt verketteten Liste. Jeder Knoten enthält das Element selbst sowie Verweise auf das vorherige und das nächste Element in der Sequenz. Sie implementiert ebenfalls das `List`-Interface sowie das `Deque`-Interface (Double-Ended Queue), siehe `Deque` (Double-Ended Queue).

12.3.1 Anwendungsbereiche

`LinkedLists` eignen sich besonders gut, wenn häufig Elemente am Anfang oder Ende der Liste hinzugefügt oder entfernt werden müssen. Sie sind daher eine gute Wahl für die Implementierung von Stacks (Stapelspeicher (Stack), LIFO (Last-In, First-Out)) oder Queues (Warteschlangen, FIFO (First-In, First-Out)).

12.3.2 Vorteile

- Sehr schnelles Einfügen und Löschen von Elementen am Anfang und Ende der Liste ($O(1)$).
- Dynamische Größenanpassung ohne die Notwendigkeit, große Speicherblöcke am Stück zu reservieren oder umzukopieren.

12.3.3 Nachteile

- Der Zugriff auf ein Element anhand seines Indexes (z.B. mit `get(index)`) ist langsam ($O(n)$), da die Liste vom Anfang (oder Ende, je nachdem, was näher ist) bis zum gewünschten Element durchlaufen werden muss.
- Benötigt mehr Speicher pro Element als eine `ArrayList`, da für jeden Knoten zusätzliche Referenzen auf Vorgänger und Nachfolger gespeichert werden müssen.

12.3.4 Beispiel

Listing 12.3: Beispiel für die Verwendung einer `LinkedList` in Java

```

1 | import java.util.LinkedList; // Import notwendig
2 | import java.util.List;      // Interface verwenden
3 |
4 | List<String> queueList = new LinkedList<>(); // Neue,  ↔
   |     leere LinkedList erstellen
5 | // Für Queue/Deque-Operationen kann man auch direkt  ↔
   |     LinkedList oder Deque verwenden:
6 | // Deque<String> queue = new LinkedList<>();
7 |
8 | queueList.add("Erster"); // Element am Ende anfügen

```



```

9  ((LinkedList<String>) queueList).addFirst("Neuer_Erster" ←
    ); // Spezifische LinkedList-Methode
10 ((LinkedList<String>) queueList).offer("Letzter"); // ←
    Typisch für Queues (fügt am Ende hinzu)
11
12 System.out.println(queueList.get(1)); // Element am ←
    Index 1 ausgeben (Ausgabe: "Erster")
13 System.out.println(((LinkedList<String>) queueList).poll ←
    ()); // Erstes Element abrufen und entfernen (Ausgabe ←
    : "Neuer Erster")

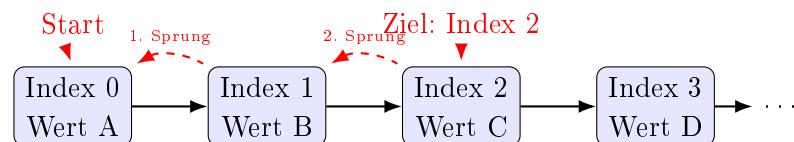
```

12.3.5 Effizienz des Indexzugriffs bei LinkedLists (möglicherweise prüfungsrelevant)

Der Zugriff auf ein Element an einem bestimmten Index (z.B. mittels `liste.get(index)`) ist bei einer `LinkedList` vergleichsweise langsam und hat eine Zeitkomplexität von $O(n)$ im schlechtesten und durchschnittlichen Fall. Der Grund hierfür liegt in der Struktur der verketteten Liste:

- Anders als bei einem Array, wo die Speicheradresse eines Elements direkt aus dem Index berechnet werden kann, muss bei einer `LinkedList` die Kette von Verweisen vom ersten (oder letzten, je nach Implementierung und Index) Knoten bis zum gewünschten Knoten verfolgt werden.
- Jeder Knoten kennt nur seinen direkten Vorgänger und Nachfolger. Um zum i -ten Element zu gelangen, müssen i Schritte (oder $n-i$ Schritte vom Ende) durchlaufen werden.

Die nachfolgende Abbildung 12.1 verdeutlicht diesen Prozess.



Zugriff auf Index 2: Es müssen 2 Sprünge vom Start (Index 0) erfolgen, um das Element zu erreichen. Jeder Sprung folgt dem Verweis des aktuellen Knotens auf den nächsten Knoten.

Abbildung 12.1: Visualisierung des sequenziellen Zugriffs auf ein Element (hier Index 2) in einer `LinkedList`. Um zum Zielknoten zu gelangen, muss die Liste vom Anfangsknoten aus durchlaufen werden.

Im Gegensatz dazu ist das sequentielle Durchlaufen aller Elemente einer `LinkedList` mit einem Iterator (z.B. in einer `for-each`-Schleife) effizient. Jeder Schritt des Iterators zum nächsten Element dauert $O(1)$, sodass das Iterieren über die gesamte Liste eine Gesamtkomplexität von $O(n)$ hat. Problematisch ist also nicht das Iterieren an sich, sondern der wahlfreie Zugriff

per Index.

12.4 HashSet

Ein `HashSet` implementiert das `Set`-Interface und speichert eine Sammlung von eindeutigen Elementen. Die Elemente in einem `HashSet` haben keine garantierte Reihenfolge; die Iterationsreihenfolge kann sich sogar ändern, wenn neue Elemente hinzugefügt werden. Die Implementierung basiert auf einer Hashtabelle (intern wird oft eine `HashMap` verwendet, bei der die Werte ignoriert werden).

12.4.1 Anwendungsbereiche

`HashSets` eignen sich hervorragend, wenn:

- Eindeutige Elemente gespeichert werden sollen und die Reihenfolge keine Rolle spielt.
- Schnell geprüft werden muss, ob ein Element bereits in der Sammlung vorhanden ist.
- Duplikate automatisch verhindert werden sollen.

12.4.2 Vorteile

- Sehr schnelle durchschnittliche Zeitkomplexität von $O(1)$ (amortisiert) für die Operationen `add`, `remove` und `contains`.
- Effiziente Vermeidung von Duplikaten.

12.4.3 Nachteile

- Keine garantierte Reihenfolge der Elemente. Die Reihenfolge bei der Iteration ist nicht vorhersagbar.
- Die Leistung kann bei einer schlechten Implementierung der `hashCode()`-Methode der gespeicherten Objekte oder bei einer sehr hohen Füllrate der internen Hashtabelle (viele Kollisionen) im schlechtesten Fall auf $O(n)$ für einzelne Operationen abfallen.

12.4.4 Beispiel

Listing 12.4: Beispiel für die Verwendung eines `HashSet` in Java

```
1 import java.util.HashSet;
2 import java.util.Set; // Interface verwenden
3
4 Set<String> uniqueNames = new HashSet<>();
5 uniqueNames.add("Alice");
6 uniqueNames.add("Bob");
7 uniqueNames.add("Alice"); // Wird ignoriert, da Duplikat ←
   (add() gibt false zurück)
8
9 System.out.println(uniqueNames.contains("Alice")); // ←
   true
```

```

10 System.out.println(uniqueNames.size()); // 2
11
12 for (String name : uniqueNames) {
13     System.out.println(name); // Reihenfolge nicht ↔
14     // garantiert, z.B. Bob, Alice
15 }

```

12.5 TreeSet

Ein `TreeSet` implementiert das `SortedSet`-Interface (welches `Set` erweitert) und speichert eine Sammlung von eindeutigen Elementen in sortierter Reihenfolge. Die Sortierung erfolgt entweder natürlich (wenn die Elemente `Comparable` implementieren) oder durch einen beim Erstellen des `TreeSet`s übergebenen `Comparator`. Die Implementierung basiert typischerweise auf einem Rot-Schwarz-Baum.

12.5.1 Anwendungsbereiche

`TreeSet`s sind nützlich, wenn:

- Eindeutige Elemente in einer sortierten Reihenfolge gespeichert und abgerufen werden müssen.
- Operationen wie das Finden des kleinsten/größten Elements oder das Abrufen von Teilmengen (Ranges) basierend auf der Sortierung benötigt werden.

12.5.2 Vorteile

- Elemente werden automatisch sortiert gehalten.
- Effiziente Operationen ($O(\log n)$) für `add`, `remove` und `contains`.
- Ermöglicht den Zugriff auf das erste/letzte Element sowie auf Teilmengen in $O(\log n)$ oder $O(1)$ (für erste/letzte).

12.5.3 Nachteile

- Langsamer als `HashSet` für die grundlegenden Operationen `add`, `remove` und `contains`, da die Sortierreihenfolge aufrechterhalten werden muss.
- Erfordert, dass Elemente entweder `Comparable` implementieren oder ein `Comparator` bereitgestellt wird. `null`-Elemente sind standardmäßig nicht erlaubt.

12.5.4 Beispiel

Listing 12.5: Beispiel für die Verwendung eines `TreeSet` in Java

```

1 import java.util.TreeSet;
2 import java.util.Set; // Interface verwenden
3 import java.util.SortedSet; // Spezifischeres Interface
4
5 SortedSet<Integer> sortedNumbers = new TreeSet<>();

```

```

6 | sortedNumbers.add(50);
7 | sortedNumbers.add(10);
8 | sortedNumbers.add(90);
9 | sortedNumbers.add(10); // Wird ignoriert, da Duplikat
10
11 | // Iteration erfolgt in sortierter Reihenfolge
12 | for (Integer number : sortedNumbers) {
13 |     System.out.println(number); // Ausgabe: 10, 50, 90
14 | }
15
16 | System.out.println(sortedNumbers.first()); // 10
17 | System.out.println(sortedNumbers.last()); // 90
18 | // System.out.println(sortedNumbers.higher(50)); // gäbe ←
    es nicht für SortedSet, nur für NavigableSet/TreeSet
19 | System.out.println(((TreeSet<Integer>)sortedNumbers). ←
    higher(50)); // 90 (Casting auf TreeSet)

```

12.6 HashMap

Eine HashMap implementiert das Map-Interface und speichert Schlüssel-Wert-Paare. Jedem Schlüssel kann höchstens ein Wert zugeordnet sein. HashMap erlaubt null-Werte und einen einzelnen null-Schlüssel. Die Reihenfolge der Einträge in einer HashMap ist nicht garantiert und kann sich ändern. Die Implementierung basiert auf einer Hashtabelle.

12.6.1 Anwendungsbereiche

HashMaps sind die Standardwahl für Map-Implementierungen, wenn:

- Daten als Schlüssel-Wert-Paare gespeichert werden sollen.
- Ein schneller Zugriff auf Werte über ihre Schlüssel erforderlich ist.
- Die Reihenfolge der Einträge keine Rolle spielt.

12.6.2 Vorteile

- Sehr schnelle durchschnittliche Zeitkomplexität von $O(1)$ (amortisiert) für die Operationen put, get, remove und containsKey.
- Flexibel durch die Erlaubnis eines null-Schlüssels und beliebig vieler null-Werte.

12.6.3 Nachteile

- Keine garantierte Reihenfolge der Schlüssel oder Werte bei der Iteration.
- Ähnlich wie bei HashSet kann die Leistung bei schlechten hashCode()-Implementierungen der Schlüssel oder hoher Füllrate auf $O(n)$ für einzelne Operationen abfallen.

12.6.4 Beispiel

Listing 12.6: Beispiel für die Verwendung einer HashMap in Java

```
1 import java.util.HashMap;
2 import java.util.Map; // Interface verwenden
3
4 Map<String, Integer> ageMap = new HashMap<>();
5 ageMap.put("Alice", 30);
6 ageMap.put("Bob", 25);
7 ageMap.put("Charlie", 35);
8 ageMap.put("Alice", 32); // Wert für "Alice" wird ↵
    aktualisiert
9
10 System.out.println(ageMap.get("Bob")); // 25
11 System.out.println(ageMap.containsKey("David")); // ↵
    false
12 ageMap.put(null, 40); // Null-Schlüssel ist erlaubt
13 ageMap.put("Eve", null); // Null-Wert ist erlaubt
14
15 for (Map.Entry<String, Integer> entry : ageMap.entrySet() ↵
    ()) {
16     System.out.println(entry.getKey() + ":_:" + entry. ↵
        getValue()); // Reihenfolge nicht garantiert
17 }
```

12.6.5 Prüfungshinweis

Dass die HashMap (und hashCode) Effizient funktioniert, muss die hashCode ↵ Methode (und damit die equals Methode) sinnvoll überschrieben werden.

12.7 TreeMap

Eine TreeMap implementiert das SortedMap-Interface (welches Map erweitert) und speichert Schlüssel-Wert-Paare in sortierter Reihenfolge der Schlüssel. Die Sortierung der Schlüssel erfolgt entweder natürlich (wenn die Schlüssel Comparable implementieren) oder durch einen beim Erstellen der TreeMap übergebenen Comparator. TreeMap erlaubt keine null-Schlüssel (im Gegensatz zu HashMap), aber null-Werte sind erlaubt. Die Implementierung basiert typischerweise auf einem Rot-Schwarz-Baum.

12.7.1 Anwendungsbereiche

TreeMaps werden verwendet, wenn:

- Schlüssel-Wert-Paare gespeichert werden müssen und eine Iteration über die Schlüssel in sortierter Reihenfolge erforderlich ist.
- Operationen benötigt werden, die von der Sortierung der Schlüssel abhängen, wie das Finden des Eintrags mit dem kleinsten/größten Schlüssel oder das Abrufen von Teilmengen basierend auf Schlüsselbereichen.

12.7.2 Vorteile

- Schlüssel werden automatisch sortiert gehalten.
- Effiziente Operationen ($O(\log n)$) für put, get, remove und containsKey.
- Ermöglicht den Zugriff auf den ersten/letzten Eintrag sowie auf Teil-Maps (Ranges) in $O(\log n)$ oder $O(1)$.

12.7.3 Nachteile

- Langsamer als HashMap für die grundlegenden Operationen, da die Sortierreihenfolge der Schlüssel aufrechterhalten werden muss.
- Erfordert, dass Schlüssel entweder Comparable implementieren oder ein Comparator bereitgestellt wird.
- Erlaubt keine null-Schlüssel (wirft NullPointerException).

12.7.4 Beispiel

Listing 12.7: Beispiel für die Verwendung einer TreeMap in Java

```
1 import java.util.TreeMap;
2 import java.util.Map; // Interface verwenden
3 import java.util.SortedMap; // Spezifischeres Interface
4
5 SortedMap<Integer, String> sortedUserMap = new TreeMap <-
    <>();
6 // Alternativ: Map<Integer, String> sortedUserMap = new <-
    TreeMap<>();
7
8 sortedUserMap.put(102, "Alice");
9 sortedUserMap.put(100, "Bob");
10 sortedUserMap.put(105, "Charlie");
11 // sortedUserMap.put(null, "David"); // Würde <-
    NullPointerException werfen
12
13 // Iteration erfolgt in sortierter Reihenfolge der Schlü <-
    ssel
14 for (Map.Entry<Integer, String> entry : sortedUserMap. <-
    entrySet()) {
15     System.out.println(entry.getKey() + ":\u0020" + entry. <-
        getValue()); // 100: Bob, 102: Alice, 105: <-
        Charlie
16 }
17
18 System.out.println(sortedUserMap.firstKey()); // 100
19 // Für subMap etc. ist SortedMap ausreichend
20 SortedMap<Integer, String> subMap = sortedUserMap.subMap <-
    (100, 103); // Einträge für 100 (inkl.), 102 (inkl.), <-
    103 (exkl.)
21 System.out.println("SubMap:\u0020" + subMap);
```

12.8 Über Collections Iterieren (Prüfungshinweis)

Über Collections, abgesehen von der `ArrayList`, sollte *immer* mittels Iterator iteriert werden. Andere lösungen mit einer Zähl Variable und mehreren `.get` aufrufen funktionieren zwar, sind aber in der Praxis sehr Ineffizient und führen zu Punktabzug.

Teil II

Übungsblätter

13. Übungsblatt Klassenhierarchie und Polymorphie

13.1 Aufgabe 1

Bearbeiten Sie diese Aufgabe ohne Zuhilfenahme des Rechners. Betrachten Sie folgende Klassen A, B und C:

```
1 public class A {  
2     public A() {  
3         System.out.println("A");  
4     }  
5  
6     public A(int parameter) {  
7         System.out.println("A2");  
8     }  
9 }  
10  
11 public class B extends A {  
12 }  
13  
14 public class C extends B {  
15     public C() {  
16         System.out.println("C");  
17     }  
18  
19     public C(int parameter) {  
20         System.out.println("C2");  
21     }  
22 }
```

Beantworten Sie nun folgende Fragen:

1. Welche Ausgaben (Reihenfolge beachten!) erfolgen bei Auswertung der Ausdrücke `new C()` und `new C(5)`?

Antwort:

`new C()` AC

`new C(5)` AC2

2. Lassen sich noch alle Klassen A, B und C fehlerfrei compilieren, wenn in B folgender Konstruktor hinzugefügt wird?

```
1 public B(int parameter) {  
2     System.out.println("B");  
3 }
```

Falls ja, welche Ausgabe erfolgt bei Auswertung von `new C(5)`? Falls nein, welche Klasse(n) lassen sich aus welchen Gründen nicht compilieren?

Antwort: Die Klasse C lässt sich nicht mehr compilieren.

13.2 Aufgabe 2

Bearbeiten Sie diese Aufgabe ohne Zuhilfenahme des Rechners. Gegeben seien die Klassen P, Q und R.

```
1 public class P {  
2     public String m1(P par) {  
3         return "[Pm1" + par.m2(this) + " ]";  
4     }  
5  
6     public String m2(P par) {  
7         return "(Pm2" + this.m3() + par.m3() + " )";  
8     }  
9  
10    public String m3() {  
11        return "Pm3";  
12    }  
13 }  
14  
15 public class Q extends P {  
16     public String m1(P par) {  
17         return "[Qm1" + super.m1(par) + " ]";  
18     }  
19  
20     public String m2(P par) {  
21         return "(Qm2" + super.m2(par) + " )";  
22     }  
23  
24     public String m3() {  
25         return "Qm3";  
26     }  
27 }  
28  
29 public class R extends Q {  
30     public String m3() {  
31         return "Rm3";  
32     }  
33 }
```

Welchen Wert hat der Ausdruck `new R().m1(new Q())`?

[Qm1[Pm1(Qm2(Pm2Qm3Rm3))]]

14. Übung Klasse Object

14.1 Aufgabe 1

Gegeben sei folgende Klasse Zeitspanne, deren Objekte Zeitspannen bestehend aus Stunden, Minuten und Sekunden repräsentieren.

```
1 public class Zeitspanne {
2     private final int stunden;
3     private final int minuten;
4     private final int sekunden;
5
6     /**
7      * Erzeugt ein Objekt dieser Klasse. Die Parameter dü ↵
8      * rfen
9      * nicht negativ sind.
10     */
11     public Zeitspanne(int stunden, int minuten, int ↵
12         sekunden) {
13         this.stunden = stunden;
14         this.minuten = minuten;
15         this.sekunden = sekunden;
16     }
17 }
```

- Erweitern Sie die Klasse derart, dass Objekte der Klasse mit beliebigen Objekten auf Gleichheit getestet werden können. Ein Objekt der Klasse Zeitspanne soll nur mit Objekten derselben Klasse gleich sein, und zwar genau dann, wenn es sich real um dieselbe zeitliche Spanne handelt.

Bsp.: 2 Stunden, 17 Minuten und 10 Sekunden ist die gleiche Zeitspanne wie 0 Stunden, 135 Minuten und 130 Sekunden.

```
1 @Override
2 public boolean equals(Object obj) {
3     if (obj instanceof Zeitspanne z) {
4         return this.stunden * 3600 + this.minuten * ↵
5             60 + this.sekunden == z.stunden * 3600 ↵
6             + z.minuten * 60 + this.sekunden;
7     }
8     return false;
9 }
```

- Erweitern Sie die Klasse derart, dass durch den Methodenaufruf `System.out.print(z)` (z sei ein Ausdruck des Typs Zeitspanne) eine textuelle Darstellung des übergebenen Objekts im Format h:mm:ss ausgegeben wird. Hierbei soll der Minuten- und Sekundenanteil immer zweistellig und kleiner als 60 sein. Der Stundenanteil wird mit so vielen Stellen dargestellt, wie es erforderlich ist.

```

1 @Override
2 public String toString() {
3     int seconds = this.stunden * 3600 + this. ←
        minuten * 60 + this.sekunden;
4     int hours = seconds / 3600;
5     seconds = seconds % hours;
6     int minutes = seconds / 60;
7     seconds = seconds / minutes;
8     return "" + hours + ":" + (minutes > 10 ? "" : ←
        "0") + minutes + ":" + (seconds > 10 ? "" : ←
        "0") + seconds;
9 }

```

14.2 Aufgabe 2

Jedes der folgenden Programmstücke enthält eine equals-Methode (und es kann noch andere Methoden enthalten). Beantworten Sie für jedes Programmstück folgende Fragen.

1. Ist das Programmstück compilierbar? Falls nein, was ist der Fehler?
2. Falls es compilierbar ist, gibt es Anwendungen der Methode, die zu einem Laufzeitfehler führen?
3. Falls Anwendungen der Methode zu einem Laufzeitfehler führen, ist dies 'akzeptabel' oder sollte er durch Änderung der Methode vermieden werden?
4. Fällt Ihnen sonst noch etwas Bemerkenswertes an dem Programmstück auf?

```

1 public class A {
2     private int b;
3
4     public boolean equals(Object obj) {
5         return this.b == ((A) obj).b;
6     }
7 }

```

1. Ist das Programmstück compilierbar? Falls nein, was ist der Fehler?
Ja
2. Falls es compilierbar ist, gibt es Anwendungen der Methode, die zu einem Laufzeitfehler führen?
Ja, wenn das Object keine A-Instanz ist

3. Falls Anwendungen der Methode zu einem Laufzeitfehler führen, ist dies 'akzeptabel' oder sollte er durch Änderung der Methode vermieden werden?

Nein

4. Fällt Ihnen sonst noch etwas Bemerkenswertes an dem Programmstück auf?

-

```
1 public class B {  
2     private int b;  
3  
4     public boolean equals(B obj) {  
5         return this.b == obj.b;  
6     }  
7 }
```

1. Ist das Programmstück compilierbar? Falls nein, was ist der Fehler?

Ja

2. Falls es compilierbar ist, gibt es Anwendungen der Methode, die zu einem Laufzeitfehler führen?

wenn obj == null

3. Falls Anwendungen der Methode zu einem Laufzeitfehler führen, ist dies 'akzeptabel' oder sollte er durch Änderung der Methode vermieden werden?

Ja, die Anforderungen an die Methode sind nicht die selben wie die Anforderungen an die Equals Methode aus der Object-Klasse

4. Fällt Ihnen sonst noch etwas Bemerkenswertes an dem Programmstück auf?

Die Methode überschreibt nicht die equals Methode aus der Object-Klasse

```

1 public class C {
2     private int b;
3
4     public void equals(Object obj) {
5     }
6 }

```

1. Ist das Programmstück compilierbar? Falls nein, was ist der Fehler?
Nein, der Return Typ darf nicht geändert werden.
2. Falls es compilierbar ist, gibt es Anwendungen der Methode, die zu einem Laufzeitfehler führen?
-
3. Falls Anwendungen der Methode zu einem Laufzeitfehler führen, ist dies 'akzeptabel' oder sollte er durch Änderung der Methode vermieden werden?
-
4. Fällt Ihnen sonst noch etwas Bemerkenswertes an dem Programmstück auf?
-

```

1 public class D {
2     private String s;
3
4     public boolean equals(Object obj) {
5         return obj.hashCode() == this.hashCode();
6     }
7 }

```

1. Ist das Programmstück compilierbar? Falls nein, was ist der Fehler?
Ja
2. Falls es compilierbar ist, gibt es Anwendungen der Methode, die zu einem Laufzeitfehler führen?
wenn obj == null
3. Falls Anwendungen der Methode zu einem Laufzeitfehler führen, ist dies 'akzeptabel' oder sollte er durch Änderung der Methode vermieden werden?
Nein, weil die Anforderungen an die Equals Methode der Object-Klasse nicht erfüllt werden
4. Fällt Ihnen sonst noch etwas Bemerkenswertes an dem Programmstück auf?
Gleicher hashCode muss nicht gleiches Objekt bedeuten.

15. Übung Collection Klassen

15.1 Aufgabe 1

Gegeben sei die Klasse Point:

```
1 public class Point {
2     private int x;
3     private int y;
4
5     public Point(int x, int y) {
6         this.x = x;
7         this.y = y;
8     }
9
10    public int getX() {
11        return this.x;
12    }
13
14    public int getY() {
15        return this.y;
16    }
17 }
```

Realisieren Sie eine statische Methode

`boolean` `doesInclude(LinkedList<Point> points, Point p)` um zu prüfen, ob eine verkettete Liste ein Point-Objekt enthält, das hinsichtlich seiner Koordinaten mit dem übergebenen Punkt übereinstimmt. Sie dürfen auch den Code der Klasse Point erweitern.

```
1 public static boolean doesInclude(LinkedList<Point> ↵
2     points, Point p) {
3     return points.contains(p);
4 }
5
6 @Override
7 public boolean equals(Object obj) {
8     if (obj instanceof Point p) {
9         this.x == p.x && this.y == p.y;
10    }
11    return false;
12 }
13
14 @Override
15 public int hashCode() {
```

```

15     return this.x + this.y;
16 }

```

15.2 Aufgabe 2

Die Menge $M \subset \mathbb{N}$ sei wie folgt induktiv definiert:

1. $1 \in M$
2. falls $m \in M$, dann sind auch $2m + 1 \in M$ und $3m \in M$

Realisieren Sie eine statische Methode `ArrayList<Integer> upTo(int n)`, die aufsteigend sortiert alle Elemente der Menge M bis zum ersten Element liefert, das größer oder gleich n ist. Beispiel: `upTo(20)` liefert eine Liste mit den Elementen 1, 3, 7, 9, 15, 19 und 21.

```

1 public static ArrayList<Integer> upTo(int n) {
2     ArrayList<Integer> list = new ArrayList<>();
3     list.add(1);
4
5     int i = 0;
6
7     while (list.get(i) < n) {
8         int num = list.get(i);
9         list.add(2 * num + 1);
10        list.add(3 * num);
11        i++;
12    }
13
14    Collections.sort(list);
15
16    while (list.size() > 1 && list.get(list.size() - 2) <=
17           > n) {
18        list.removeLast();
19    }
20
21    list = new ArrayList<>(new TreeSet<>(list));
22
23    return list;
24 }

```

15.3 Aufgabe 3

Diese Aufgabe bezieht sich auf die praktische Aufgabe ‘Car Sharing’. Ein `VehicleManager` verwaltet viele Fahrzeuge an vielen Standorten, ein Fahrzeug kann für viele Zeiträume gebucht werden. Überlegen Sie, wie man diese Daten der Car Sharing-Anwendung am besten repräsentieren kann. Berücksichtigen Sie insbesondere, welche Zugriffe auf diese Daten aufgrund der Methoden der Klasse `VehicleManager` erforderlich sind, und denken Sie an Laufzeiteffizienz.

HashMap.

15.4 Aufgabe 4

Skizzieren Sie die notwendigen Erweiterungen der Klasse Fraction (Praktikum zu EPR), damit ein Objekt der Klasse HashSet<Fraction> keine zwei Brüche mit gleichem Wert enthalten kann.

```
1  @Override
2  public boolean equals(Object obj) {
3      if (obj instanceof Fraction f) {
4          return this.numerator == f.numerator && this.denominator == f.denominator;
5      }
6      return false;
7  }
8
9  @Override
10 public int hashCode() {
11     return this.numerator + this.denominator;
12 }
```

15.5 Aufgabe 5

Diese Aufgabe können Sie erst bearbeiten, wenn Sie das Kapitel ‘Hüllklassen’ durchgearbeitet haben. Ein Objekt der Klasse Zeitgeschichte soll Ereignisse der Zeitgeschichte verwalten. Welche Instanzvariablen wählen Sie (es kommt vor allem auf die Typen an), sodass eine Instanzmethode gibEreignisse (int jahr) möglichst effizient die Ereignisse eines Jahres liefert? Skizzieren Sie den Quellcode dieser Methode, ebenso wie den Quellcode des Konstruktors sowie einer Methode void fuegeEreignisHinzu(int jahr, String ereignis).

Nimmt man ein TreeMap<Integer, String> als Collection, so sind die Ereignisse sehr effizient gespeichert und sie sind direkt sortiert.

```
1  public class Zeitgeschichte {
2      TreeMap<Integer, Set<String>> ereignisse;
3
4      public Zeitgeschichte() {
5          this.ereignisse = new TreeMap<>();
6      }
7
8      public Set<String> gibEreignis(int jahr) {
9          return ereignisse.get(jahr);
10     }
11
12     public void fuegeEreignisHinzu(int jahr, String ereignis) {
13         Set<String> set = ereignisse.getOrDefault(jahr, new HashSet<>());
14         set.add(ereignis);
15         ereignisse.put(jahr, set);
16     }
17 }
```

15.6 Aufgabe 6

Diese Aufgabe bezieht sich auf die praktische Aufgabe ‘Staff’. Realisieren Sie in der Klasse `Employee` eine statische Methode

`...getHierarchy(ArrayList<Employee>)`, die aus den übergebenen Mitarbeitern und allen Vorgesetzten, auf die darüber zugegriffen werden kann, eine Personalararchie erstellt. Eine passende Datenstruktur dafür müssen Sie selbst finden, weshalb der Ergebnistyp der Methode offen gelassen ist. In der Personalararchie sollen alle Personen enthalten sein, die oben genannt sind, und jeder Person sollen ihre direkt untergebenen Personen zugeordnet sein. Für diese Aufgabe ist es zulässig, dass verschiedene Mitarbeiter den gleichen Namen haben. Anders ausgedrückt: Haben zwei Objekte der Klasse `Employee` und `Superior` verschiedene Identitäten, handelt es sich um verschiedene Personen. Diese Aufgabe ist ein Beispiel dafür, dass man nicht zwingend `hashCode` und `equals` in einer Klasse überschreiben muss, wenn man Objekte dieser Klasse als Schlüssel in einer `HashMap` verwendet. Warum nicht? Weil in `Object` beide Methoden implementiert sind - und manchmal ist die Implementierung dort genau die, die man benötigt.

```
1  HashMap<Superior, HashSet<Employee>> getHierarchy( ↵
    ArrayList<Employee> employees) {
2      HashMap<Superior, HashSet<Employee>> hierarchie = ↵
          new HashMap<>();
3
4      for (Employee employee : employees) {
5          Employee superior = employee.getSuperior();
6
7          HashSet<Employee> list = hierarchie.getOrDefault ↵
              (superior, new HashSet<>());
8          list.add(employee);
9
10         hierarchie.put(superior, list);
11     }
12
13     return hierarchie;
14 }
```

16. Übung Ausnahmen

16.1 Aufgabe 1

Ist es durch Code-Änderungen innerhalb der Klasse Fraction (Praktikumsaufgabe EPR) möglich, dafür zu sorgen, dass sich keine Objekte dieser Klasse mit Nenner 0 erzeugen lassen? Wenn nein, warum nicht? Wenn ja, wie?

Ja, indem eine Exception (wie z.B. eine `IllegalArgumentException`) geworfen wird, wenn dies der Fall ist.

16.2 Aufgabe 2

Realisieren Sie eine auf JUnit basierende Testmethode für folgenden Testfall:

1. Eine `ArrayList` für ganze Zahlen als Komponenten erzeugen (passenden Typparameter selbst wählen).
2. Der Liste zwei beliebige Werte hinzufügen.
3. Mittels `get`-Methode versuchen, auf den Wert an Indexposition 3 zuzugreifen.
4. Für diesen Testablauf wird erwartet, dass eine Ausnahme der Klasse `IndexOutOfBoundsException` mit der Meldung `Index 3 out of bounds for length 2` geworfen wird.

Sie können eines der beiden Realisierungsschemas verwenden, das wir in der Vorlesung kennengelernt haben, um das Auftreten erwarteter Ausnahmen zu testen. Das Schema, das auf JUnit 5 basiert, ist zwar noch ein bisschen geheimnisvoll, aber Sie kommen anhand des Code-Beispiels aus der Vorlesung bestimmt damit zurecht.

```
1 @Test
2 void testArrayList() {
3     ArrayList<Integer> list = new ArrayList<>(List.of(1, ↵
4         2));
5     IndexOutOfBoundsException e = assertThrows( ↵
6         IndexOutOfBoundsException.class, () -> list.get ↵
7         (3));
8     assertEquals(e.getMessage(), "Index_3_out_of_bounds_ ↵
9         for_length_2");
10 }
```

16.3 Aufgabe 3

An welchen Stellen innerhalb der Klasse Parser (Praktikumsaufgabe ‘Arithmetic Expression’) ist es sinnvoll, eine ParseException zu werfen, die einen Fehler in einem zu parsenden Ausdruck anzeigt?

Immer dann wenn ein ungültiges Token gefunden wird.

17. Übung Schnittstellen

17.1 Aufgabe 1

Ist folgendes Code-Stück korrekt? Was ist ggfs. der Compile- oder Laufzeitfehler?

```
1 public class C {  
2     private Map<String, String> eineMap;  
3  
4     public C() {  
5         eineMap = new HashMap<>();  
6     }  
7  
8     public HashMap<String, String> m(String s) {  
9         eineMap.put(s, s);  
10        return eineMap;  
11    }  
12 }
```

Das Code-Stück führt zu einem Compile-Fehler. Der Rückgabewert der Methode `m` ist eine `HashMap`, die Instanzvariable `eineMap` ist allerdings eine `Map`. Eine `HashMap` ist zwar eine `Map`, aber eine `Map` ist nicht zwingend `HashMap`.

17.2 Aufgabe 2

Gegeben seien folgende Klassen:

```
1 public class Kreis {
2     private double radius;
3
4     public Kreis(double radius) {
5         this.radius = radius;
6     }
7
8     public double gibUmfang() {
9         return 2 * radius * Math.PI;
10    }
11 }
12
13 public class Rechteck {
14     private double laenge;
15     private double breite;
16
17     public Rechteck(double laenge, double breite) {
18         this.laenge = laenge;
19         this.breite = breite;
20     }
21 }
```

Definieren Sie eine Schnittstelle Geo und ergänzen Sie die oben stehenden Klassen, sodass folgender Programmcode compiliert und mit dem gewünschten Verhalten ausgeführt werden kann.

```
1 Geo g = new Kreis(2.0);
2 g = new Rechteck(2.0, 1.0);
3
4 // Figur in alle Richtungen um Faktor 3 vergrößern
5 g.skaliere(3);
6
7 // erwartete Ausgabe: Umfang = 18.0
8 System.out.println("Umfang_=" + g.gibUmfang())
```

```
1 public interface Geo {
2     void skaliere(int i);
3     double gibUmfang();
4 }
5
6 public class Kreis implements Geo {
7     private double radius;
8
9     public Kreis(double radius) {
10         this.radius = radius;
11     }
12
13     public double gibUmfang() {
14         return 2 * radius * Math.PI;
15     }
16 }
```

```

17     public void skaliere(int i) {
18         this.radius *= i
19     }
20 }
21
22 public class Rechteck implements Geo{
23     private double laenge;
24     private double breite;
25
26     public Rechteck(double laenge, double breite) {
27         this.laenge = laenge;
28         this.breite = breite;
29     }
30     public double gibUmfang() {
31         return 2 * laenge + 2 * breite;
32     }
33
34     public void skaliere(int i) {
35         this.laenge *= i;
36         this.breite *= i;
37     }
38 }

```

17.3 Aufgabe 3

Realisieren Sie eine auf JUnit basierende Testmethode für die Methode `getHierarchy` aus der Übung zu Collection-Klassen. Da wir mittlerweile Schnittstellen kennengelernt haben, gehen wir von folgender Definition der Methode aus:

```

1  /**
2   * Liefert aus den übergebenen Employees und allen
3   * Superiors, auf die darüber zugegriffen werden
4   * kann, eine Personalhierarchie. In der
5   * Personalhierarchie sind alle Personen
6   * enthalten, die oben genannt sind, und jeder
7   * Person sind ihre direkt untergebenen Personen
8   * zugeordnet.
9   *
10  * @param employees Mitarbeiter, von denen ausgehend
11  * die Hierarchieinformation ermittelt wird.
12  *
13  * @return die Personalhierarchie basierend auf den
14  * übergebenen Mitarbeitern. Die Schlüssel
15  * der Zuordnung sind alle direkten und indirekten
16  * Vorgesetzten dieser Mitarbeiter.
17  * Jedem Vorgesetzten sind als Wert die direkten
18  * Untergebenen zugeordnet.
19  */
20 Map<Superior, Set<Employee>> getHierarchy(Collection<↵
    Employee> employees)

```

```

1 void testGetHierarchie() {
2     Superior kriegesmann = new Superior("kriegesmann");
3     Superior luis = new Superior("luis");
4     Employee urban = new Employee("urban");
5     Employee borsum = new Employee("borsum");
6     luis.setSuperior(kriegesmann);
7     urban.setSuperior(luis);
8     borsum.setSuperior(luis);
9
10    HashMap<Superior, ArrayList<Employee>> hierarchie = ↵
        getHierarchy(List.of(kriegesmann, luis, urban, ↵
            borsum));
11
12    HashMap<Superior, ArrayList<Employee>> ↵
        expectedHierarchie = new HashMap<>();
13    expectedHierarchie.put(kriegesmann, List.of(luis));
14    expectedHierarchie.put(luis, List.of(urban, borsum)) ↵
        ;
15    expectedHierarchie.put(urban, new ArrayList<>());
16    expectedHierarchie.put(borsum, new ArrayList<>());
17
18    assertEquals(expectedHierarchie, hierarchie);
19 }

```

17.4 Aufgabe 4

In welcher Klasse (oder welchen Klassen) der praktischen Aufgabe Number Sequence kann sinnvoll die Klasse PushBackSequence eingesetzt werden? Erläutern Sie, zu welchem Zweck die Klasse dort eingesetzt wird.

MergeSequence und UniqueSequence

17.5 Aufgabe 5

An welchen Stellen innerhalb der Klassen der praktischen Aufgabe Number Sequence ist es erforderlich oder sinnvoll, eine NoSuchElementException zu erzeugen und zu werfen?

in Allen endlichen NumberSequences bei der `.getNext()` Methode.

18. Übung Funktionale Schnittstellen

18.1 Aufgabe 1

Schreiben Sie ein Code-Stück, um ein Objekt der Klasse `TreeSet< Point>` zu erzeugen, bei dem die Punkte nach der folgenden Regel geordnet sind. Am Ende des Code-Stücks soll die Menge über die Variable `points` zugreifbar sein. Gehen Sie davon aus, dass es eine geeignete Klasse `Point` gibt.

Sind `p` und `q` zwei Elemente der Menge, dann liegt `p` vor `q`, wenn gilt: Die `x`-Koordinate von `p` ist kleiner als die `x`-Koordinate von `q`, oder die `x`-Koordinaten beider Punkte sind gleich und die `y`-Koordinate von `p` ist kleiner als die `y`-Koordinate von `q`.

```
1 TreeSet<Point> points = new TreeSet<>((p, q) -> p.getX() <=
    - q.getY() == 0? p.getX() - q.getY() : p.getY() - q.
    getY());
```

18.2 Aufgabe 2

Diese Aufgabe bezieht sich auf das Code-Beispiel „UserManagement“ der Vorlesung. Definieren Sie die Schnittstelle `PasswordValidator` derart, dass bei der Erzeugung eines `UserManagement`-Objekts ein `PasswordValidator` per Lambda-Ausdruck übergeben werden kann. Erzeugen Sie auf diese Weise ein `UserManagement`-Objekt, das Passwörter akzeptiert, die mit einer Ziffer beginnen und mindestens die Länge 12 haben.

```
1 new UserManager(s -> s.charAt(0) >= '0' && s.charAt(0) <=
    <= '9' && p.length() >= 12);
```

19. Übung Streams

19.1 Aufgabe 1

Gegeben sei folgende Methode. Schreiben Sie die Methode unter Verwendung von Streams derart um, dass darin weder eine explizite Iteration, noch eine if-Anweisung vorkommt

```
1 public static List<String> m1(Collection<String> woerter ↵
   ) {
2     List<String> ergebnis = new ArrayList<>();
3     for (String wort : woerter) {
4         if (wort.length() >= 5) {
5             String s = wort.substring(wort.length() / 2);
6             if (s.charAt(0) >= 'a') {
7                 ergebnis.add(s);
8             }
9         }
10    }
11    return ergebnis;
12 }
```

19.2 Aufgabe 2

Schauen Sie sich die API-Dokumentation der Klasse OptionalInt, insbesondere der Methode ifPresent an. Ändern Sie dann das folgende Code-Stück derart, dass es bei gleichem Verhalten keine if-Anweisung mehr enthält. Die Variable is ist vom Typ IntStream.

```
1 OptionalInt zahlOpt = is
2     .filter(n -> n % 2 == 0)
3     .findAny();
4 if(zahlOpt.
5
6 isPresent()){
7     int zahl = zahlOpt.getAsInt();
8     System.out.
9
10 println(zahl *zahl);
11 }
```

19.3 Aufgabe 3

Schauen Sie sich die API-Dokumentation der Schnittstelle `LongStream`, insbesondere der Methode `takeWhile` an. Geben Sie unter Verwendung der Methode `LongStream fibonaccis()` aus dem Code-Beispiel der Vorlesung einen Ausdruck an, der den Stream aller Fibonacci-Zahlen liefert, die mathematisch korrekt berechnet werden (d. h. bei denen es zu keinen Überlauf kommt).

19.4 Aufgabe 4

Es sei `s` eine Variable des Typs `Set<Integer>`.

1. Geben Sie einen Ausdruck an, der die Summe aller Elemente der Menge `s` berechnet, die geradzahlig sind.
2. Geben Sie einen Ausdruck des Typs `IntStream` an, der alle natürlichen Zahlen enthält bis einschließlich 100, die nicht in der Menge `s` enthalten sind.

19.5 Aufgabe 5

Geben Sie einen Ausdruck des Typs `IntStream` an, der folgende Elemente enthält: 0, 1, 3, 7, 15, 31, ...

19.6 Aufgabe 6

Realisieren Sie eine Methode `boolean istPalindrom(String s)` ohne Verwendung von Schleifen, if-Anweisungen und Rekursion.

19.7 Aufgabe 7

Gegeben sei das folgende Code-Stück? Die Variable `zeilen` ist vom Typ `Stream<String>`.

```
1 zeilen
2     .map(w -> new StringTokenizer(w))
3     .filter(st -> st.hasMoreTokens())
4     .map(st -> st.nextToken())
5     .filter(s -> s.length() >= 5)
6     .findFirst();
```

1. Was berechnet der Ausdruck?
2. Jemand behauptet, das Code-Stück sei völlig ineffizient und würde, wenn `zeilen` einen un- endlichen Stream enthält, überhaupt nicht terminieren. Er begründet dies damit, dass durch die erste Anwendung von `map` für alle Elemente des Streams zunächst ein `StringTokenizer` erzeugt wird. Ändern Sie das Code-Stück derart, dass durch Bildschirmausgaben dokumentiert wird, dass die Person Unrecht hat und nur soviele `StringTokenizer` erzeugt werden, wie für die Funktion des Code-Stücks benötigt werden.

Glossar

Call Stack (Aufrufstapel / Aufrufkamin) Der Call Stack ist eine dynamische Datenstruktur, die vom Laufzeitsystem eines Programms verwendet wird, um den Überblick über die aktiven Methodenaufrufe zu behalten. Man kann ihn sich als einen Stapel von Blöcken vorstellen, bei dem jeder Block (ein sogenannter *Stack Frame*) Informationen zu einem einzelnen Methodenaufruf enthält. Wenn eine Methode aufgerufen wird, wird ein neuer Frame für diese Methode oben auf den Stack gelegt. Dieser Frame beinhaltet unter anderem die lokalen Variablen der Methode und die Rücksprungsadresse (die Stelle im Code, zu der nach Beendigung der Methode zurückgekehrt werden soll). Wenn eine Methode ihre Ausführung beendet, wird ihr Frame vom obersten Ende des Stacks entfernt, und die Kontrolle kehrt zur aufrufenden Methode zurück (anhand der Rücksprungsadresse). Tritt eine Exception auf und wird nicht behandelt, wird der Call Stack verwendet, um den Stack Trace zu generieren. 36, 39

Deque (Double-Ended Queue) Eine Deque (ausgesprochen *Deck*, kurz für Double-Ended Queue) ist eine Datenstruktur, die das Einfügen und Entfernen von Elementen an beiden Enden (Anfang und Ende) effizient erlaubt. Sie kann somit sowohl als Warteschlange (Queue, FIFO) als auch als Stapelspeicher (Stack, LIFO) verwendet werden. 25

FIFO (First-In, First-Out) Das FIFO-Prinzip (engl. First-In, First-Out) besagt, dass das Element, das zuerst zu einer Struktur hinzugefügt wurde, auch als erstes wieder entnommen wird. Man kann es sich wie eine Warteschlange vorstellen: Wer zuerst kommt, mahlt zuerst. 25

LIFO (Last-In, First-Out) Das LIFO-Prinzip (engl. Last-In, First-Out) besagt, dass das Element, das zuletzt zu einer Struktur hinzugefügt wurde, als erstes wieder entnommen wird. Dies ist typisch für einen Stapelspeicher (Stack). 25

Optional Die Klasse `Optional` umhüllt ein anderes Objekt, ähnlich wie die Klasse `Integer` einen `int`-Wert umhüllt. Der Wert eines `Optional` kann mit der Instanzmethode `get` bekommen werden. Hat das `Optional` kein Element, so wirft dies eine `NoSuchElementException`. Um zu überprüfen, ob das `Optional` einen Wert beinhaltet, kann die Instanzmethode

`isPresent` benutzt werden. Die Klasse `Optional` hat noch ein paar wenige weitere Funktionen, allerdings wurden diese in der Vorlesung nicht besprochen. Mehr hier:
<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html>. 46

Stack Trace (Aufrufliste / Stapelrückverfolgung) Ein Stack Trace ist eine detaillierte Liste der Methodenaufrufe, die zu einem bestimmten Zeitpunkt im Programm aktiv waren; typischerweise genau in dem Moment, als eine Exception ausgelöst und nicht sofort behandelt wurde. Er zeigt die Kette der Methodenaufrufe in umgekehrter Reihenfolge an: von der Methode, in der die Exception direkt aufgetreten ist, zurück bis zur Methode, die den Prozess gestartet hat (oft die `main`-Methode). Jeder Eintrag im Stack Trace repräsentiert einen Methodenaufruf und enthält üblicherweise den Klassennamen, den Methodennamen und die Zeilennummer im Quellcode, an der der Aufruf zur nächsten Methode erfolgte oder die Exception auftrat. Der Stack Trace ist ein unverzichtbares Werkzeug für das Debugging, da er Entwicklern hilft, den genauen Pfad und Kontext zu verstehen, der zu einem Fehler geführt hat. 37, 39, 40

Stapelspeicher (Stack) Ein Stapelspeicher, auch Stack genannt, ist eine abstrakte Datenstruktur, die nach dem LIFO-Prinzip (Last-In, First-Out) arbeitet. Elemente werden oben auf den Stapel gelegt (`push`) und auch von oben wieder entfernt (`pop`). 25

Vollständig Qualifiziert Pakete und Klassen, welche über ihren vollen Namen angesprochen werden, nennt man *vollständig Qualifiziert*. So wird also anstelle von `Date` `java.util.Date` verwendet. Klassen müssen vollständig Qualifiziert werden, wenn mehrere Klassen gleich heißen. Beispielsweise `java.util.Date` und `java.sql.Date`. 3