Westfälische Hochschule

OPR

Alles wichtige auf einen Blick

OPR Semesterzusammenfassung

Author(s):
-

Supervisor(s):
Prof. Luis

2. Juli 2025

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

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: $p_1.p_2.\ldots.p_n$. 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: $p_1.p_2.\ldots.p_n.K$, 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 ↩
           Zuweisung
10     // Date ambiguDate; // Fehler: Mehrdeutigkeit ohne ↩
           Qualifizierung
11   }
12 }
```

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 ↩
        verfügbar
6   }
7 }
```

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  ↩
          Unterklasse
3     // oder überschriebene Methoden der Oberklasse
4 }
```

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

```
29      // der deklarierte Typ der Referenz (Car) für die  ↩
            Bindung verwendet.
30      // System.out.println(meinFahrzeug.getCategorie());  ↩
            // Schlechter Stil! Sollte Vehicle.getCategorie() ↩
             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␣ ↩
            aufgerufen␣mit:␣" + name);
6     }
7     public BaseClass() {
8       this.name = "Default";
9       System.out.println("Parameterloser␣Konstruktor␣ ↩
            Basisklasse␣aufgerufen");
10    }
11  }
12
13  class ChildClass extends BaseClass {
14    public ChildClass(String spezifischerName) {
15      super(spezifischerName); // Ruft BaseClass(String  ↩
            name) auf
16      System.out.println("Konstruktor␣ChildClass␣ ↩
            aufgerufen");
17    }
18    public ChildClass() {
19      super(); // Ruft BaseClass() auf
20      // Alternativ: super("DefaultAbgeleitet");
```

```
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 ërraten".

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.methodenName()` 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: " +
           benutzername + ", Aktiv: " + istAktiv);
11   }
12
13   public Benutzer(String benutzername) {
14     this(benutzername, benutzername + "@example.com");
           // Ruft den ersten Konstruktor auf
15     System.out.println("Benutzer (nur Name) erstellt, E-
           Mail generiert.");
16   }
17
18   public Benutzer() {
19     this("gast"); // Ruft den zweiten Konstruktor auf,
           der dann den ersten aufruft
20     this.istAktiv = false; // Überschreibt den
           Standardwert für Gast-Benutzer
21     System.out.println("Gast-Benutzer erstellt, Inaktiv
           gesetzt.");
22   }
23 }
24
25 public class TestThisKonstruktor {
26   public static void main(String[] args) {
27     Benutzer b1 = new Benutzer("alice", "alice@mail.com"
           );
28     System.out.println("---");
29     Benutzer b2 = new Benutzer("bob");
30     System.out.println("---");
31     Benutzer b3 = new Benutzer();
32   }
33 }
```

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 ↩
           ndert␣werden.");
4    }
5    public void andereOperation() {
6      System.out.println("Diese␣Operation␣kann␣ü ↩
           berschrieben␣werden.");
7    }
8  }
9
10 class AbgeleitetVonFinal extends BasisMitFinal {
11   // @Override
12   // public final void wichtigeOperation() { } //  ↩
         KOMPILIERFEHLER!
13
14   @Override
15   public void andereOperation() {
16     System.out.println("andereOperation␣in␣ ↩
           AbgeleitetVonFinal␣überschrieben.");
17   }
18 }
```

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.

3. JUnit Tests (Sehr Prüfungsrelevant)

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 aufgerufen.
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 ↩
       , die gemockt werden könnte
4
5    public String processData() {
6      String data = fetchData();
7      return "Processed: " + data.toUpperCase();
8    }
9  }
10
11 // Eine konkrete Implementierung der abstrakten Klasse f ↩
     ür Testzwecke (eine Art Mock)
12 class TestableDataProcessor extends  ↩
     AbstractDataProcessor {
13   private String dataToReturn;
14
15   public TestableDataProcessor(String dataToReturn) {
16     this.dataToReturn = dataToReturn;
17   }
18
19   @Override
20   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 mit kontrollierten Daten
31      AbstractDataProcessor processor = new TestableDataProcessor("sample data");
32      String result = processor.processData();
33      assertEquals("Processed: SAMPLE DATA", result);
34    }
35  }
```

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 gehöhren 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 addresse zurück.

4.0.2 toString

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

Die Klasse Objekt hat noch viele weitere Methoden implementiert, allerdings sind keien 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
```

```
numbers[3] = 42; // Dem Element am Index 3 den Wert 42 zuweisen
System.out.println(numbers[3]); // Wert am Index 3 ausgeben (Ausgabe: 42)
// 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

```
import java.util.ArrayList; // Import notwendig
import java.util.List; // Interface verwenden ist gute Praxis

List<Integer> list = new ArrayList<>(); // Neue, leere ArrayList erstellen
list.add(10); // Element 10 am Ende hinzufügen
list.add(20); // Element 20 am Ende hinzufügen
list.add(0, 5); // Element 5 am Index 0 einfügen ( verschiebt andere Elemente)

```

```
System.out.println(list.get(1)); // Element am Index 1 ausgeben (Ausgabe: 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

```
import java.util.LinkedList; // Import notwendig
import java.util.List;       // Interface verwenden

List<String> queueList = new LinkedList<>(); // Neue, leere LinkedList erstellen
// Für Queue/Deque-Operationen kann man auch direkt LinkedList oder Deque verwenden:
// Deque<String> queue = new LinkedList<>();

queueList.add("Erster"); // Element am Ende anfügen
```

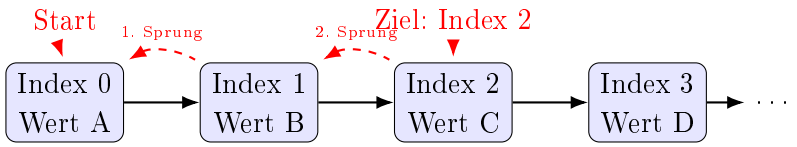
```
9  ((LinkedList<String>) queueList).addFirst("Neuer Erster" ↩
       ); // Spezifische LinkedList-Methode
10 ((LinkedList<String>) queueList).offer("Letzter"); //  ↩
       Typisch für Queues (fügt am Ende hinzu)
11
12 System.out.println(queueList.get(1)); // Element am  ↩
       Index 1 ausgeben (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 5.1 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
```

```
System.out.println(uniqueNames.size()); // 2

for (String name : uniqueNames) {
    System.out.println(name); // Reihenfolge nicht garantiert, z.B. Bob, Alice
}
```

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 `TreeSets` übergebenen `Comparator`. Die Implementierung basiert typischerweise auf einem Rot-Schwarz-Baum.

5.5.1 Anwendungsbereiche

`TreeSets` 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

```
import java.util.TreeSet;
import java.util.Set; // Interface verwenden
import java.util.SortedSet; // Spezifischeres Interface

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

```java
import java.util.HashMap;
import java.util.Map; // Interface verwenden

Map<String, Integer> ageMap = new HashMap<>();
ageMap.put("Alice", 30);
ageMap.put("Bob", 25);
ageMap.put("Charlie", 35);
ageMap.put("Alice", 32); // Wert für "Alice" wird aktualisiert

System.out.println(ageMap.get("Bob")); // 25
System.out.println(ageMap.containsKey("David")); // false
ageMap.put(null, 40); // Null-Schlüssel ist erlaubt
ageMap.put("Eve", null); // Null-Wert ist erlaubt

for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue()); // Reihenfolge nicht garantiert
}
```

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() + ":␣" + 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:␣" + 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 äbstürzt".

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 Exception auslöst
3      // z.B. int result = 10 / 0; // Würde eine ArithmeticException auslösen
4      // oder
5      // String text = null;
6      // System.out.println(text.length()); // Würde eine NullPointerException auslösen
7  } catch (SpecificExceptionType1 e1) {
8      // Behandlung für SpecificExceptionType1
9  }
10 // ... 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: " + ↩
           e.getMessage());
8  } finally {
9      if (reader != null) {
10         try {
11             reader.close(); // Wichtig: Ressourcen ↩
                   freigeben!
12         } catch (java.io.IOException e) {
13             System.err.println("Fehler beim Schliessen ↩
                   des Readers: " + e.getMessage());
14         }
15     }
16     System.out.println("Finally-Block wurde ausgeführt." ↩
           );
17 }
```

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 ↩ (new String[])`
- Über die Instanzmethode der Klasse `BufferedReader`: `BufferedReader ↩ 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(Comperator<? super T> comperator)` und `max(Comperator<? super T> comperator)` Gibt den Maximalen bzw Minimalen Wert zugrunde des Comperators. Comperator 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 alelr byteorientierten Eingabeströme. Im prinzip gibt es 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 sein. Jedoch kann der Ergebniswert auch -1 sein, falls das Byte nicht existiert. 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 IntStream

```
1 IntStream is;
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 einen `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. Aufgrunddessen 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.

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 Rücksprungadresse (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 Rücksprungadresse). 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 beinhält, 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