Programmieren 2 - Fortgeschrittene Programmierung

Matthias Berg-Neels

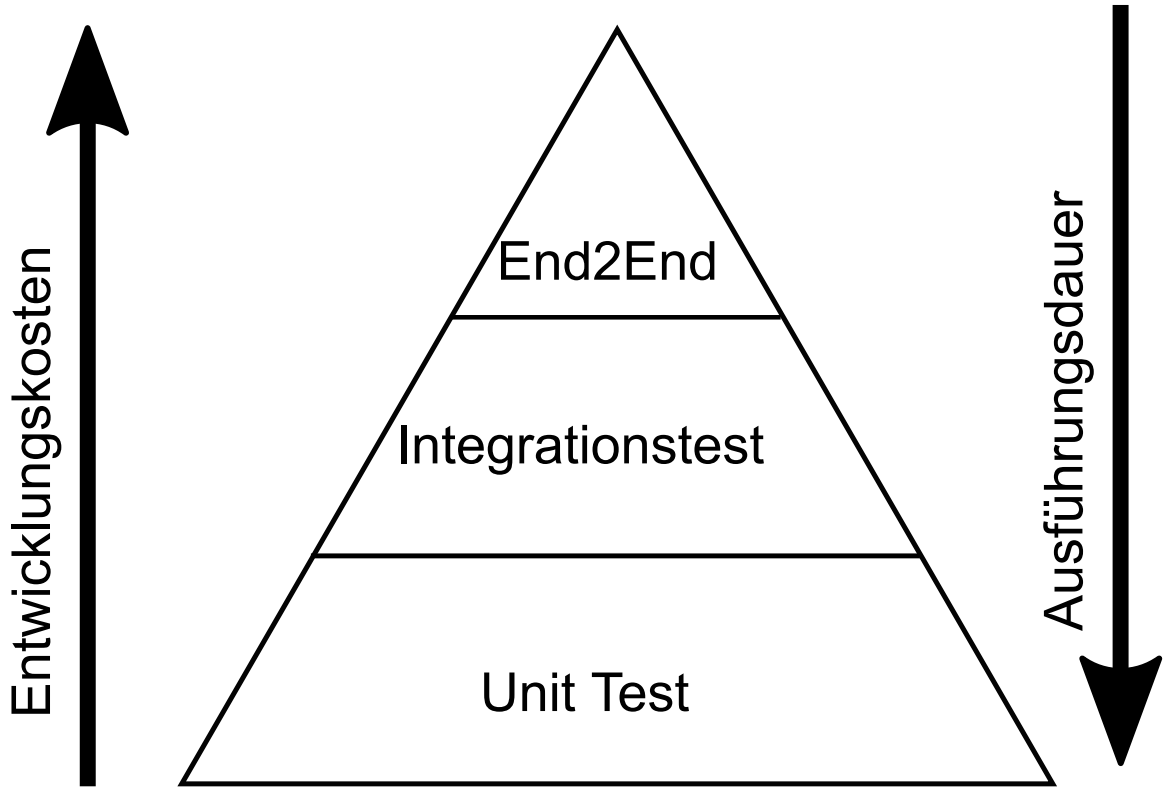
Kapitelübersicht - Programmieren 2

8. Exception Handling
9. Collection Framework
10. Swing
11. Optional: Input- & Output-Stream
12. Optional: Threads

Exkurs
Unit Testing

Unit Testing in Java mit JUnit5.

Einordnung von Unit Tests



JUnit5 - Annotations

Annotation	Beschreibung
`@Test`	kennzeichnet eine Methode als Test
`@Displayname("<Name>")`	definiert den Anzeigename für den jeweiligen Test / die Testklasse
`@Nested`	markiert eine innere (geschachtelte) Testklasse
`@Tag("<tag>")`	definiert einen Tag zur Filterung von Tests
`@BeforeEach`	kennzeichnet eine Methode die **vor jedem** Test läuft ((!)JUnit4 --> `@Before`)
`@AfterEach`	kennzeichnet eine Methode die **nach jedem** Test läuft ((!)JUnit4 --> `@After`)
`@BeforeAll`	kennzeichnet eine Methode die **einmal vor allen** Tests läuft ((!)JUnit4 --> `@BeforeClass`)
`@AfterAll`	kennzeichnet eine Methode die **einmal nach allen** Tests läuft ((!)JUnit4 --> `@AfterClass`)

Assertion

- Zusicherung / Sicherstellung / Assertion (lat. Aussage / Behauptung)
  - Definition einer Erwartungshaltung zum Vergleich gegen den tatsächlichen Zustand
- JUnit Tests:
  - Klasse: `Assertions` (`org.junit.jupiter.api.Assertions`)
  - statische Methoden zur Definition eines erwartenden Ergebnisses (expected) zum Vergleich mit dem tatsächlichen Ergebnis (actual)
  - überladene Methoden mit zusätzlichem Parameter `Message` für eigene Meldungen
  - automatische Validierung der "Behauptung" durch das JUnit Test-Framework
  - beliebt als statischer Import zur direkten Nutzung der Methoden:

```
import static org.junit.jupiter.api.Assertions.*;
```

  - Beispiele:

```
Assertions.assertEquals(<expected>, <actual>[, <Message>]);
Assertions.assertNotEquals(<expected>, <actual>[, <Message>]);
Assertions.assertTrue(<actual>[, <Message>]);
Assertions.assertFalse(<actual>[, <Message>]);
Assertions.assertTimeout(<expected Duration>, <Executable>[, <Message>]);
Assertions.assertThrows(<expected Exception-Class>, <Executable>[, <Message>]);
```

Beispiel Test-Klasse

```java
import excercises.exkurs.junit.Calculator;
import org.junit.jupiter.api.*;

class CalculatorTest {

    Calculator myCalculator;
    double result = 0;

    @BeforeEach
    void setUp() {
        myCalculator = new Calculator();
        result = 0;
    }

    @Test
    @DisplayName("adding two numbers")
    void add() {
        result = myCalculator.add(5.0, 10.0);
        Assertions.assertEquals(15.0, result);
    }
}
```

F.I.R.S.T. Principal

- **Fast**: Die Testausführung soll schnell sein, damit man sie möglichst oft ausführen kann. Je öfter man die Tests ausführt, desto schneller bemerkt man Fehler und desto einfacher ist es, diese zu beheben.
- **Independent**: Unit-Tests sind unabhängig voneinander, damit man sie in beliebiger Reihenfolge, parallel oder einzeln ausführen kann.
- **Repeatable**: Führt man einen Unit-Test mehrfach aus, muss er immer das gleiche Ergebnis liefern.
- **Self-Validating**: Ein Unit-Test soll entweder fehlschlagen oder gut gehen. Diese Entscheidung muss der Test treffen und als Ergebnis liefern. Es dürfen keine manuellen Prüfungen nötig sein.
- **Timely**: Man soll Unit-Tests vor der Entwicklung des Produktivcodes schreiben.

Kapitel 8
Exception Handling

Kapitelübersicht - Programmieren 2

8. **Exception Handling**
9. Collection Framework
10. Swing
11. Optional: Input- & Output-Stream
12. Optional: Threads

Lernziele

- Sie kennen die unterschiedlichen Ausnahmen in Java
- Sie können eigene Ausnahmeklassen definieren
- Sie können Ausnahmen auslösen und weitergeben
- Sie können Ausnahmen behandeln und das Ausnahmenkonzept in Java erläutern
- Sie können den Unterschied zwischen ckecked und unchecked Exceptions erklären

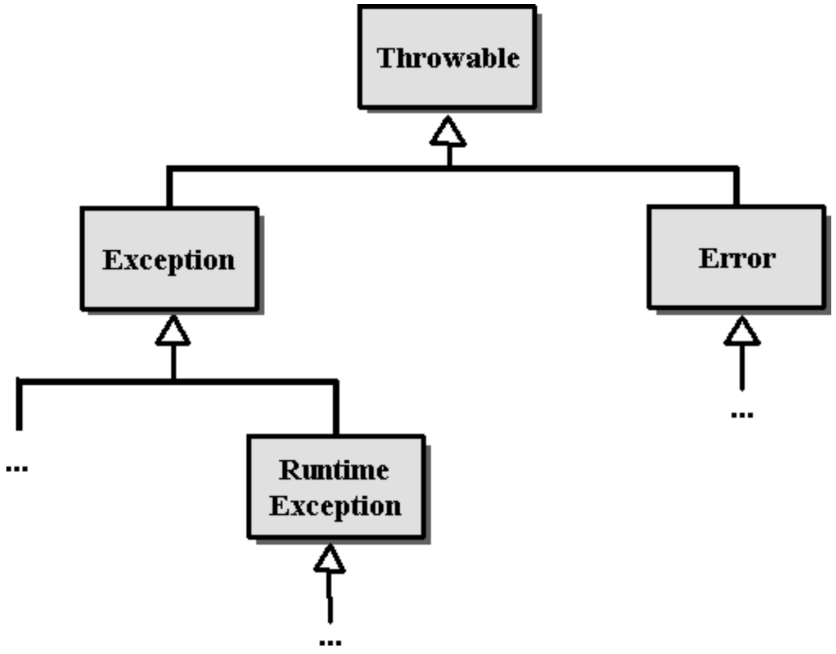
Fehler in Java

Compiler-Fehler

- syntaktische Fehler werden beim Kompilieren erkannt

Laufzeitfehler

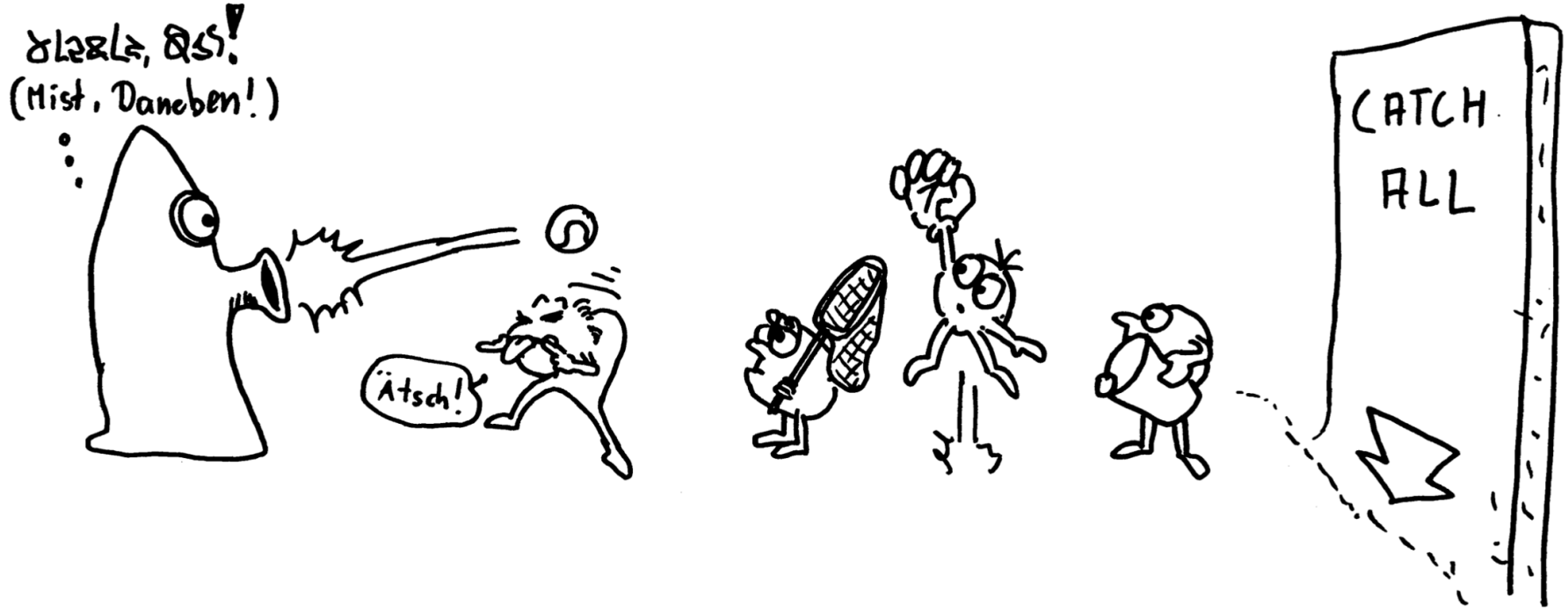
- Fehler (Error) sollte nicht behandelt werden
- Ausnahmen (Exceptions)
  - Exception muss behandelt werden
  - RuntimeException kann behandelt werden



Grundprinzip der Ausnahmebehandlung

- Laufzeitfehler oder explizite Anweisung löst Ausnahme aus
- 2 Möglichkeiten der Fehlerbehandlung
  - Direkte Fehlerbehandlung im auslösenden Programmteil
  - Weitergabe der Ausnahme an die aufrufende Methode
- bei Weitergabe liegt die Entscheidung beim Empfänger
  - Er kann die Ausnahme behandeln
  - Er kann die Ausnahme an seinen Aufrufer weitergeben
- wird die Ausnahme nicht behandelt, führt sie zur Ausgabe einer Fehlermeldung und zum Programmabbruch (Laufzeitfehler)

Ausnahmen behandeln



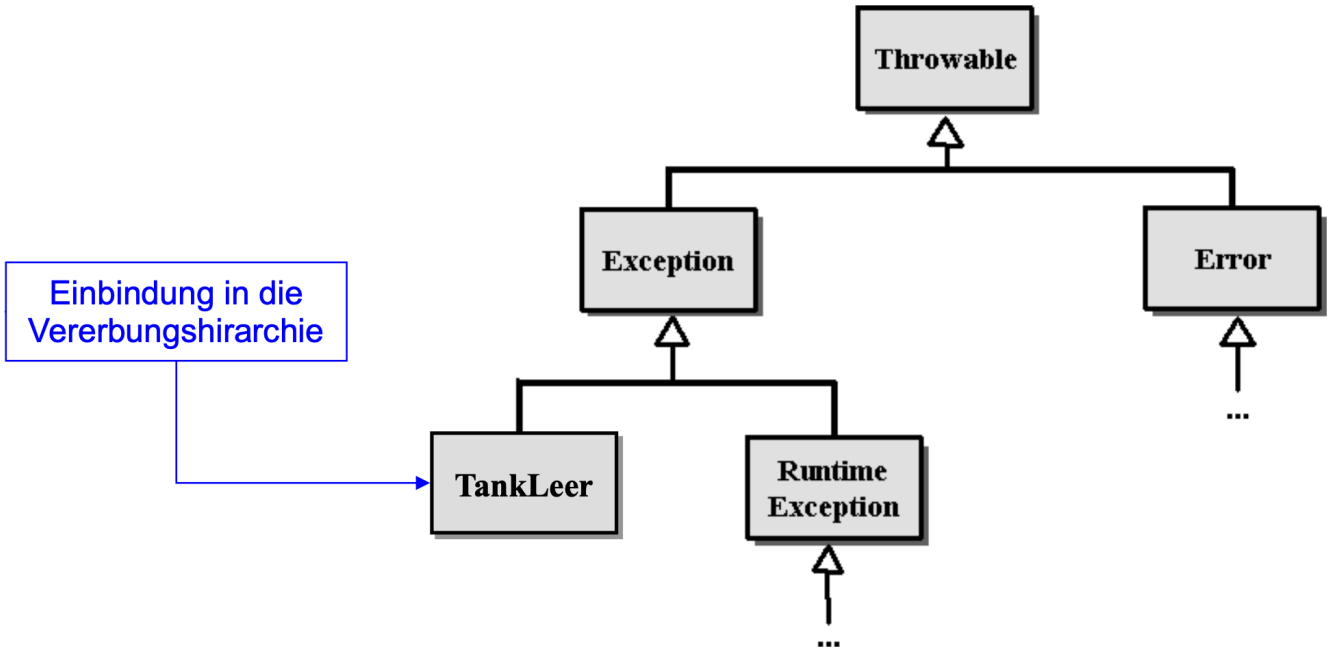
© Christian Ullenboom,Java ist auch eine Insel, 3.Auflage, S. 359

- Überwachung des Codingbereichs, in dem Ausnahmen ausgelöst werden können
- spezieller Code zur Behandlung aufgetretener Ausnahmen

Eigene Ausnahmeklassen

Erzeugen eigener Ausnahmeklassen

```
public class TankLeer extends Exception {
  public TankLeer (int km) {
    super("Der Tank ist nach " + km + " Kilometern leer.");
  }
}
```



Ausnahmen explizit auslösen und weitergeben

```
public class Auto {
  // ...
  public void fahren() throws TankLeer {
    while (true) {
      if (fuel > 0) { fuel -= 6;
        tagesKM += 100;
        kmCount += 100;
      } else {
          throw new TankLeer(tagesKM);
      }
    }
  }
  //...
}
```

- Definition der möglichen Ausnahmen in Methoden-Signatur: `throws`
- Erzeugen eines neuen Ausnahme-Objektes: `new TankLeer(tagesKM)`
- Auslösen (werfen) der Ausnahme (im Ausnahmefall) innerhalb der Methode: `throw`

Ausnahme behandeln

```
public class TankLeerDemo {
  public static void main(String[] args) {
    Auto bmw = new Auto(0, 35487);
    //...
    try {
      bmw.fahren();
    } catch (TankLeer e1) {
      System.out.println(e1.getMessage());
      System.out.println(e1.toString()); e1.printStackTrace();
    } catch (Exception e2) {
      e2.printStackTrace();
    }
    // ...
  finally {
    System.out.println("Der neue Kilometerstand: " + bmw.getKmCount());
  }
  //...
 }
}
```

- `try` markiert den Überwachungsbereich - ausgelöste Ausnahmen beenden die Ausführung des Überwachungsbereiches umgehend
- `catch` fängt mögliche Ausnahmen aus dem Überwachungsbereich auf
- `finally` wird unabhängig vom Auftreten von Ausnahmen ausgeführt

Wichtige Methoden der Klasse Throwable

`public String getMessage()`

- liefert den Fehlertext zurück

```
Der Tank ist nach 1100 Kilometern leer.
```

`public String toString()`

- liefert die Objektbeschreibung und den Fehlertext zurück

```
prog2.demos.exceptions.TankLeer: Der Tank ist nach 1100 Kilometern leer.
```

`public void printStackTrace()`

- liefert die Objektbeschreibung, den Fehlertext sowie die Weitergabehierarchie bis zur genauen Auslösestelle zurück

```
prog2.demos.exceptions.TankLeer: Der Tank ist nach 1100 Kilometern leer.
at prog2.demos.exceptions.Auto.fahren(Auto.java:21)
```

Checked VS. Unchecked Exceptions

Checked

- müssen verarbeitet werden
  - abfangen mit `try / catch`
  - weiterleiten mit `throws`
- werden explizit ausgelöst `throw`

Unchecked

- treten zur Laufzeit auf (RuntimeException)
- werden automatisch an den Aufrufer weitergeben
- können abgefangen werden `try / catch`
- oftmals logische Programmfehler
  - `Division by Zero`
  - `NullPointerException`
  - `IndexOutOfBoundsException`

Ausnahmen in JUnit-Tests

- spezielle Assertion
  - Rückgabe des Ausnahme Objektes
  - schlägt fehl, wenn keine oder eine andere Ausnahme zurück geworfen wird

```
Assertions.assertThrows(<Erwartete Ausnahme Klasse>. <Executable Interface>[, <Message>
```

- Assertion für den gegenläufigen Fall
  - schlägt fehl, wenn eine Ausnahme geworfen wird

```
Assertions.assertDoesNotThrow(<Executable Interface> [, <Message>]);
```

- (!) JUnit4: spezielles Attribut in `@Test` Annotation

```
@Test(expected = <Erwartete Ausnahme Klasse>)
```

Exkurs
Innere Klassen

von inneren Klassen hin zu Lambda-Funktionen

Arten von inneren Klassen

- Innere Top-Level Klasse
  - Geschachtelte statische Klasse innerhalb einer anderen Klasse mit Bezeichner (Klassenname)
  - können innerhalb und außerhalb (abhängig von der Sichtbarkeit) der Klasse verwendet werden
- Innere Element Klasse
  - Geschachtelte Klasse innerhalb einer anderen Klasse mit Bezeichner (Klassenname)
  - können innerhalb und außerhalb (abhängig von der Sichtbarkeit) der Klasse verwendet werden
    - nur im Kontext eines Objekts der äußeren Klasse
- Innere lokale Klasse
  - Geschachtelte Klasse innerhalb einer Methode mit Bezeichner (Klassenname)
  - können nur innerhalb der Methode (Scope) genutzt werden
- Innere anonyme Klasse
  - geschachtelte Klasse innerhalb einer anderen Klasse / Methode **ohne** Bezeichner
  - werden direkt einer Referenz zugwiesen
  - basieren immer auf einer Klasse (erweitern) oder einem Interface (implementieren)

Innere Top-Level-Klasse

```
package main.inner.toplevelclass;

public class OuterClass {

    // Innerhalnb einer anderen Klasse definierte Top-Level Klasse
    public static class InnerTopLevelClass{
        void print(String printText){
            System.out.println(this.getClass().getName() + " " + printText);
        }
    }

    private static void printFromInnerTopLevelClass(String printText) {
        OuterClass.InnerTopLevelClass myInnerTopLevelClass = new OuterClass.InnerTopLevelClass();
        myInnerTopLevelClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        OuterClass.printFromInnerTopLevelClass("Inner Top-Level Class: HelloWorld");
    }
}
```

Innere Element-Klasse

```
package main.inner.elementclass;

public class OuterClass {

    // Innerhalb einer andere Klasse definierte Element Klasse
    public class InnerElementClass {
        void print(String printText){
            System.out.println(this.getClass().getName() + " " + printText);
        }
    }

    void printFromInnerElementClass(String printText){
        OuterClass.InnerElementClass myInnerElementClass = this.new InnerElementClass();

        myInnerElementClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromInnerElementClass("Inner Element Class: HelloWorld");
    }
}
```

Innere lokale Klasse

```
package main.inner.local;

public class OuterClass {

    void printFromLocalInnerClass(String printText){
        // innerhalb einer Methode (Scope) definierte Klasse
        class LocalInnerClass{
            void print(String printText){
                System.out.println(this.getClass().getName() + " " + printText);
            }
        }

        LocalInnerClass myLocalInnerClass = new LocalInnerClass();

        myLocalInnerClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromLocalInnerClass("local inner Class: HelloWorld");
    }
}
```

Innere anonyme Klasse

```java
package main.inner.anonym;

public class OuterClass {

    private static interface Printable{
        void print(String printText);
    }

    void printFromAnonymousInnerClass(String printText) {
        // ohne eigenen Bezeichner definiert (kann nicht wiederverwendet werden)
        // erweitert eine bestehende Klasse oder implementiert ein Interface
        OuterClass.Printable myAnonymousInnerClass = new OuterClass.Printable() {
            @Override
            public void print(String printText) {
                System.out.println(this.getClass().getName() + " " + printText);
            }
        };

        myAnonymousInnerClass.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromAnonymousInnerClass("Inner anonymous Class: HelloWorld");
    }
}
```

Lambda Funktionen (anonyme Funktionen)

- seit Java 8
- reine Funktionen ohne eigene Klasse
- Definition: `()->{}`
- implementieren ein funktionales Interface (Interface mit **einer** Methode)
  - ersetzen (unter dieser Voraussetzung) anonyme Klassen
- haben Zugriff auf den umliegenden Kontext (finale / effektiv finale Variablen)
  - in diesem Zusammenhang auch als "Closure" bezeichnet
- verkürzte Schreibweise durch Herleitung der Informationen aus Interface-Definition
- werden an eine Referenz übergeben (direkt oder indirekt)

```
Interface1 lambda1 = parameter -> Anweisung;
Interface2 lambda2 = (parameter1, parameter2) -> Anweisung;
Interface3 lambda3 = () -> {
  Anweisung1;
  Anweisung2;
  Anweisung3;
}
```

Lambda Funktion

```
package main.lambda;

public class OuterClass {

    private static interface Printable{
        void print(String printText);
    }

    void printFromLambdaFunction(String printText) {
        // Lambda Funktionen sind "reine Funktionen" ohne Klasse
        // nutzen immer ein funktionales Interface (nur eine Methode)
        // zur Implementierung
        OuterClass.Printable myLambdaPrintFunction = (lambdaPrintText) -> {
            System.out.println(this.getClass().getName() + " " + lambdaPrintText);
        };

        myLambdaPrintFunction.print(printText);
    }

    public static void main(String[] args) {
        OuterClass myClass = new OuterClass();

        System.out.println("OuterClass: " + myClass.getClass().getName());
        myClass.printFromLambdaFunction("Lambda Function: HelloWorld");
    }
}
```

Kapitel 9
Collection Framework

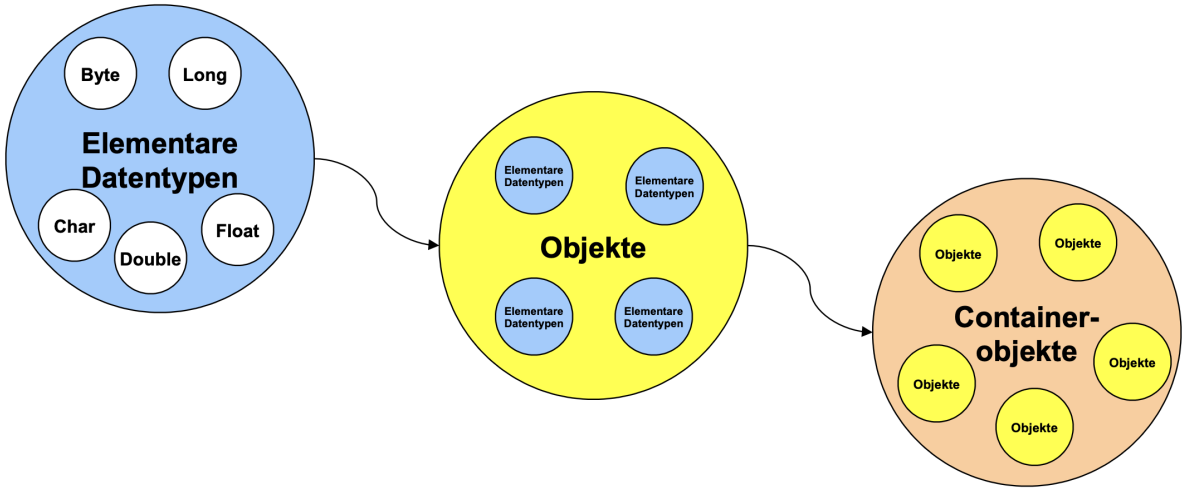
Kapitelübersicht - Programmieren 2

8. Exception Handling
9. **Collection Framework**
10. Swing
11. Optional: Input- & Output-Stream
12. Optional: Threads

Lernziele

- Sie können die Unterschiede der 3 Objekt-Containerarten erklären
- Sie können Objekte in den Containern einfügen, löschen und finden
- Sie können mit Iteratoren die Container durchlaufen
- Sie können sortierbare Container mit Comparable und Comparator sortieren
- Sie können die `equals()` und die `hashCode()` Methode in eigenen Klassen überschreiben
- Sie können den Zusammenhang zwischen den Methoden `equals()`, `hashCode()` und `compareTo()` erklären

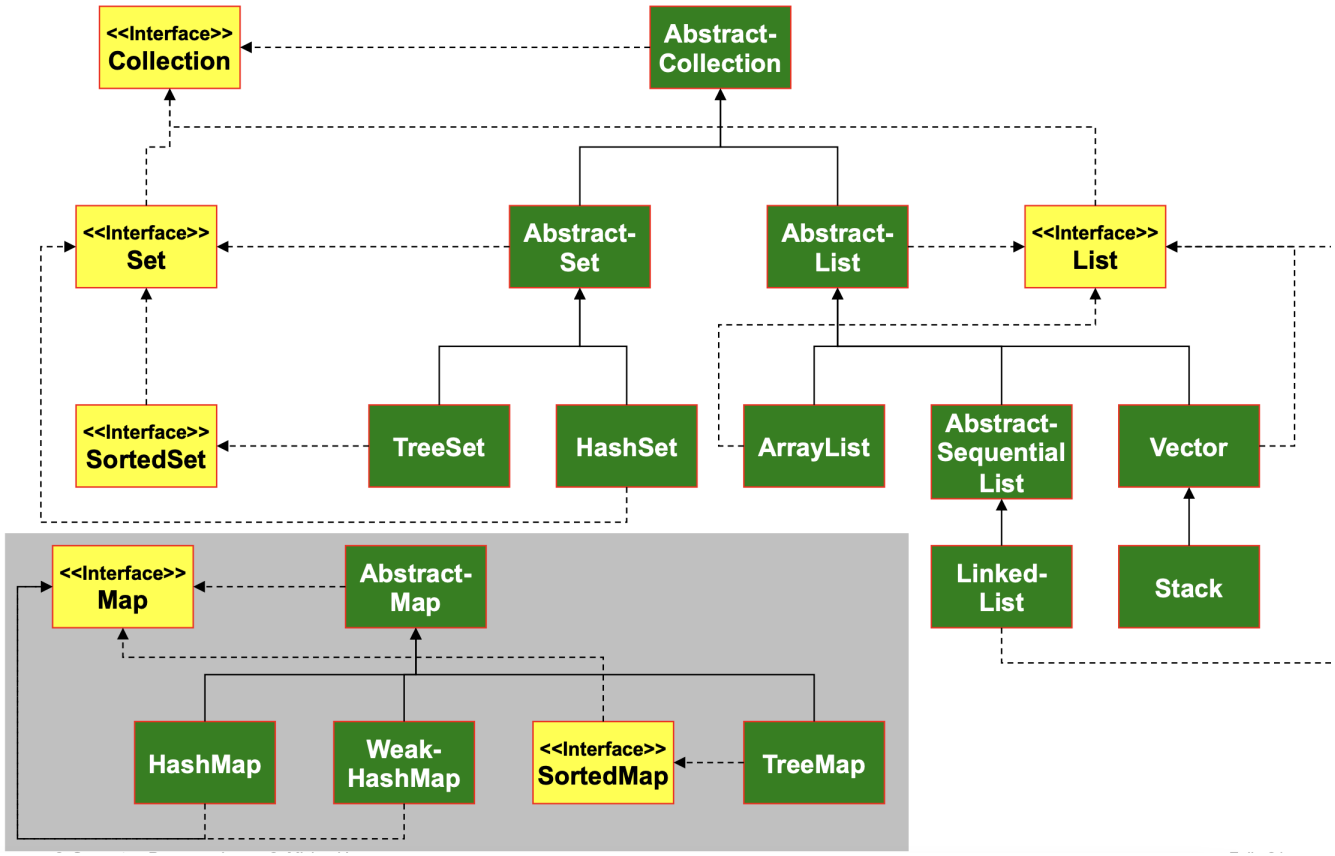
Datenstrukturen und -container



Das Collection Framework bietet generische Container

- können verschiedenste Objekte enthalten
- können beliebig viele Objekte aufnehmen
- können auf bestimmte Objekte typisiert werden

Überblick über das Collection Framework



Die drei Arten von Containern

Listen (List)

- Zugriff sequentiell oder wahlfrei
- Duplikate erlaubt
- Reihenfolge des Einfügens bleibt erhalten

Mengen (Set)

- Zugriff erfolgt über Iteratoren
- keine Duplikate
- Reihenfolge des Einfügens bleibt nicht erhalten

Schlüssel-Werte-Paare (Map)

- zusammengehörige Objektpaare
- Schlüssel sind immer eindeutig
- Zugriff über Schlüssel

Listen

List

Das Interface `List`

- befindet sich im Package java.util
- Zugriff auf die Container erfolgt sequentiell oder über
- Indexzugriff
- sequentieller Zugriff erfolgt über Iteratoren
- Index beginnt mit 0 und endet bei n Elementen bei n-1
- Größe der Liste wird dynamisch beim Einfügen oder Löschen von Elementen angepasst
- Duplikate sind erlaubt
- die Reihenfolge, in der Elemente eingefügt werden, bleibt erhalten
- meist genutzte Implementierung: `ArrayList` & `Vector`
  - intern als Arrays realisiert
  - Hauptunterschied zwischen ArrayList und Vector: Zugriffsmethoden auf Vector sind synchronisiert (wichtig bei Threads)

Wesentliche Methoden im Umgang mit Listen

- `add(int i, Object o)` oder `add(Object o)` fügt neue Objekte in die Liste ein
- `set(int i, Object o)` überschreibt das Objekt an der Stelle i mit dem Objekt o
- `get(int i)` liefert das Objekt an der Stelle i zurück
- `contains(Object o)` überprüft, ob das Objekt o in der Liste enthalten ist
- `indexOf(Object o)` liefert den Index zurück, an der das Objekt o in der Liste abgelegt ist (-1, wenn das Objekt nicht enthalten ist)
- `remove(int i)` oder remove(Object o) löscht das Objekt aus der Liste
- `clear()` initialisiert die Liste
- `size()` liefert die Länge der Liste zurück

Der Umgang mit Iteratoren

Merkmale von Iteratoren

- einheitlicher Standard zum Durchlaufen von Datencontainern l Container wird sequentiell durchlaufen
- es können keine Elemente übersprungen werden
- der Container kann sowohl vorwärts als auch rückwärts durchlaufen werden
- bei Änderung des Containerinhalts muss der Iterator neu erzeugt werden

Wichtige Iterator-Methoden

- `hasNext()` überprüft, ob das aktuelle Element im Container noch einen Nachfolger hat
- `next()` greift auf das nächste Element des Containers zu
- `remove()` löscht das Element aus dem Container, welches zuletzt vom Iterator gelesen wurde

Beispiel für eine List mit Iteratoren

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import prog2.demos.exceptions.Auto;

public class ListDemo {
    public static void main(String[] args) {
        List myList = new ArrayList();
        myList.add("Otto");
        myList.add("Karl");
        myList.add("Ludwig");
        myList.add(new Auto(0, 0));
        myList.add(2,"Otto");
        myList.set(3,"Überschreibt den Ludwig");

        System.out.println(myList.contains("Otto"));
        System.out.println(myList.indexOf("Ludwig"));
        System.out.println(myList.get(3));
        System.out.println(myList.size());

        Iterator i = myList.iterator(); while (i.hasNext()) {
            System.out.println(i.next());
        }

        myList.clear();
        System.out.println(myList.size());
    }
}
```

Mengen

Set

Die Klasse TreeSet

- befindet sich im Package java.util
- Zugriff auf die Container erfolgt sequentiell über Iteratoren
- Index beginnt mit 0 und endet bei n Elementen bei n-1
- Größe der Liste wird dynamisch beim Einfügen oder Löschen von Elementen angepasst
- Duplikate sind nicht erlaubt (Vergleich über die equals-Methode) I die Reihenfolge, in der Elemente eingefügt werden, bleibt nicht erhalten
- Einfluss auf die Sortierung der Elemente
- Sortieren nach der natürlichen Ordnung durch Implementierung des Comparable-Interface
  - Das Comparable-Interface muss auf jeden Fall implementiert werden, wenn Objekte in ein TreeSet eingefügt werden
  - Beliebige Sortierung durch Implementierung des Comparator-Interface

Beispiel für eine Menge mit Iteratoren

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Iterator;

public class SetDemo {
    public static void main(String[] args) {
        Set mySet = new TreeSet();
        mySet.add("Otto");
        mySet.add("Karl");
        mySet.add("Ludwig");

        System.out.println(mySet.contains("Otto"));
        System.out.println(mySet.size());

        Iterator i = mySet.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }

        mySet.clear();
        System.out.println(mySet.size());
    }
}
```

Ordnung und Sortierung von Objekten

Das Interface `Comparable`

- sortiert Elemente beim Einfügen in Sets oder Maps
- Sortierung erfolgt nach der natürlichen Ordnung
- muss für alle Klassen implementiert werden, deren Instanzen in Sets oder Maps gespeichert werden
- beinhaltet genau eine Methode: public int `compareTo(Object o)`
- Bedeutung der Rückgabewert
  - `Wert < 0`: das einzufügende Element liegt vor dem Vergleichsobjekt
  - `Wert = 0`: das einzufügende Element und das Vergleichsobjekt sind gleich
  - `Wert > 0`: das einzufügende Element liegt hinter dem Vergleichsobjekt

Beispiel für eine Comparable-Implementierung

```
public class Student implements Comparable {
  private String vorname;
  private String nachname;
  private int matrikelNo;

  public Student(String vorname, String name, int matrikelNo) {
    this.vorname = vorname;
    this.nachname = name;
    this.matrikelNo = matrikelNo;
  }

  // ...

  public int compareTo(Object vStudent) {
    return this.matrikelNo - ((Student) vStudent).getMatrikelNo();
  }
}
```

Beispiel TreeSet mit eigener Comparable-Implementierung

```
import java.util.Iterator;
import java.util.TreeSet;

public class DemoMenge1 {

  public static void main(String[] args) {
    TreeSet menge = new TreeSet();
    menge.add(new Student("Peter", "Maier", 75382));
    menge.add(new Student("Hans", "Müller", 65871));
    menge.add(new Student("Karl", "Schmidt", 19853));
    menge.add(new Student("Hans", "Müller", 65872));
    menge.add(new Student("Karl", "Schmidt", 19853));

    Iterator i = menge.iterator();
    while(i.hasNext()) {
      Student studie = (Student) i.next();
      System.out.println(studie.getMatrikelNo() + " " +
        studie.getVorname() + " " + studie.getNachname());
    }
  }
}
```

Das Interface `Comparator`

- sortiert Elemente beim Einfügen in Sets oder Maps
- Sortierung erfolgt nach einer beliebigen Sortierreihenolge und übersteuert die natürliche Ordnung
- Comparator sollten in eigener Klasse implementiert werden
- zur Verwendung des Comparators wird die implementierende Klasse dem Konstruktor des Sets oder der Map übergeben
- beinhaltet genau eine Methode: `public int compare(Object o1, Object o2)`
- Beeutung der Rückgabewerte
  - `Wert < 0`: o1 liegt vor o2
  - `Wert = 0`: o1 und o2 sind gleich
  - `Wert > 0`: o1 liegt hinter o2

Beispiel für eine Comparator-Implementierung

```
import java.util.Comparator;

public class StudentComparator implements Comparator{

  public int compare(Object obj1, Object obj2) {
    Student studie1 = (Student) obj1;
    Student studie2 = (Student) obj2;
    if ((studie1.getNachname().compareTo(studie2.getNachname())) != 0) {
      return studie1.getNachname().compareTo(studie2.getNachname());
    } else if ((studie1.getVorname().compareTo(studie2.getVorname())) != 0) {
      return studie1.getVorname().compareTo(studie2.getVorname());
    } else if ((studie1.getMatrikelNo() - studie2.getMatrikelNo()) != 0) {
      return studie1.getMatrikelNo() - studie2.getMatrikelNo();
    }

    return 0;
  }
}
```

Beispiel TreeSet mit eigener Comparator-Implementierung

```
import java.util.*;

public class DemoMenge1 {
  public static void main(String[] args) {
    TreeSet menge = new TreeSet(new StudentComparator());

    menge.add(new Student("Peter", "Maier", 75382));
    //...
    menge.add(new Student("Karl", "Maier", 85383));

    Iterator i = menge.iterator();
    while(i.hasNext()) {
      Student studie = (Student) i.next(); System.out.println(studie.getMatrikelNo() +
      studie.getVorname() + " " + studie.getNachname());
    }
  }
}
```

Sortieren von Listen

- Listen (`Vector`, `ArrayList`, ...) sind normalerweise unsortiert
- die Klasse `Collections` bietet eine überladene Sortiermethode zum Sortieren von List-Objekten an
- folgende Sortiermöglichkeiten werden angeboten
  - `static void sort(List liste)`
    - sortiert die Liste nach der natürlichen Ordnung
    - dazu müssen die Klassen das Interface Comparable implementieren, deren Instanzen in der Liste gespeichert sind
  - `static void sort(List liste, Comparator c)`
    - übersteuert die natürliche Ordnung und sortiert die Objekte der Liste über den entsprechenden Comparator c

Vergleichen von Objekten

equals() **und** *hashCode()*

... und *compareTo(Object o)*

Der Vergleich von Objekten

- Vergleich mit dem ==-Operator prüft, ob es sich um die identische Speicherreferenz handelt
- inhaltliche Vergleiche erfolgen über die `equals()`-Methode (`equals()`-Methode der Klasse Object entspricht dem ==-Operator)
- der equals-Contract aus der Dokumentation zur Klasse Object
  - reflexiv: jedes Objekt liefert beim Vergleich mit sich selbst true
  - symmetrisch: x verglichen mit y liefert das gleiche Ergebnis, wie der Vergleich von y mit x
  - transitiv: wenn x gleich y und y gleich z ist, dann ist auch x gleich z
  - konsistent: solange sich zwei Objekte nicht verändern, liefert der Vergleich der beiden Objekte immer das gleiche Ergebnis
  - Objekte müssen von null verschieden sein

Das Überschreiben der `equals()`-Methode

direkte Sub-Klasse von `Object`

- Alias-Check mit dem ==-Operator
- Test auf null
- Typverträglichkeit überprüft, ob es sich um Instanzen der gleichen Klasse handelt
- Feld-Vergleich überprüft die inhaltliche Gleichheit der Attribute

indirekte Sub-Klasse von `Object`

- Alias-Check mit dem ==-Operator
- Delegation an die Oberklasse ermöglicht die Prüfung der Gleichheit der von der Oberklasse geerbten Anteile
- Feld-Vergleich überprüft die inhaltliche Gleichheit der Attribute der Sub-Klasse

Das Überschreiben der `equals()`-Methode

direkte Subklasse von `Object`

```
public class Haustier {
  private String art;
  private int gewicht;
  //...

  public boolean equals(Object objekt) {
    // Alias-Check
    if (this == objekt) {
        return true;
    }
    // Test auf null
    if (objekt == null){
      return false;
    }
    // Typverträglichkeit
    if (objekt.getClass() != this.getClass()){
      return false;
    }

    // Feldvergleich
    if(!this.art.equals(((Haustier) objekt).getArt())){
      return false;
    }
    if(!(this.gewicht == ((Haustier) objekt).getGewicht())) {
      return false;
    }

    return true;
  }
}
```

Das Überschreiben der `equals()`-Methode

indirekte Subklasse von `Object`

```
public class Hund extends Haustier {
  private String rasse;
  //...

  public boolean equals(Object objekt) {
    // Alias-Check
    if (this == objekt){
        return true;
    }

    // Delegation an super
    if (!super.equals(objekt)){
      return false;
    }

    // Feldvergleich
    if (!this.rasse.equals(((Hund) objekt).getRasse())){
        return false;
    }

    return true;
  }
}
```

Zusammenhang `hashCode()` und `equals()`

- Verwendung für die Verwaltung der Einträge in hash-basierten Datencontainern (HashSet, HashMap, ...)
- korrekte Verwaltung der Einträge basiert auf folgender Bedingung (hashCode-Contract)
  - wenn `o1.equals(o2)` den Wert `true` liefert,
  - dann muss `o1.hashCode()` den gleichen Wert ergeben, wie `o2.hashCode()`
- sobald die `equals()`-Methode überschrieben wird, muss auch die `hashCode()`-Methode überschrieben werden, so dass o.g. Bedingung erfüllt wird
- Vorschlag zur Implementierung
  - Verwendung der Attribute, die bei der Implementierung der `equals()`-Methode verwendet werden
  - Ermittlung der Hash-Codes der ausgewählten Attribute einer Klasse
  - Addition oder bitweise Verknüpfung mit exklusivem Oder der einzelnen Hash-Codes

Überschreiben von `hashCode()`

```
public class Haustier {
  private String art;
  private int gewicht;
  //...

  // Getter- und Setter-Methoden
  public boolean equals(Object objekt) {
    //...
  }

  public int hashCode() {
    return this.getArt().hashCode() ^ this.getGewicht();
  }
}
```

```
public class Hund extends Haustier {
  private String rasse;
  //...

  public boolean equals(Object objekt) {
    //...
  }

  public int hashCode() {
    return super.hashCode() ^ this.rasse.hashCode();
  }
}
```

`hashCode()` – Alternative Implementierung

Typ	Zugeordneter Integer Wert
Boolean	`(field ? 0 : 1)`
byte, char, short, int	`(int) field`
long	`(int)(field>>>32) ^ (int)(field & 0xFFFFFFFF)`
float	`((x==0.0F) ? 0 : Float.floatToIntBits(field))`
double	`((x==0.0) ? 0L : Double.doubleToLongBits(field))` [anschliessende Behandlung wie bei long]
Referenz	`((field==null) ? 0 : field.hashCode())`

`hashCode()` – Alternative Implementierung

```
public class Haustier {
  private String art;
  private int gewicht;

  // ...
  public int hashCode() {
    int hc = 17;                // beliebiger Initialwert
    int hashMultiplier = 59;  // beliebige (kleine) Primzahl

    hc = hc * hashMultiplier + (field==null) ? 0 : field.hashCode()) + gewicht;
    return hc;
  }
}
```

Was hat das mit `Comparable`zu tun?

- `comapreTo()` sortiert Objekte nach einer "natürlichen" Ordnung
  - Rückgabe Wert 0: die Objekte sind gleich
  - damit sollte der Rückgabewert 0 für zwei Objekte einem Rückgabewert von true beim vergleich mit `equals()` entsprechenden (Comparable-Contract)

> *`equals()` und `compareTo()` sollten sich konsistent verhalten*

- Zusammengefasst: `equals()`, `hashCode()` und `compareTo()` **sollten** für ein Objekt immer auf den gleichen Attributen basieren

Schlüssel-Werte-Paare

Maps

Das Interface `Map`

- befindet sich im Package java.util
- ist kein Sub-Interface von Collection
- es werden immer Schlüssel-Werte-Paare eingefügt
- jeder Schlüssel ist eindeutig
- wird mit dem gleichen Schlüssel ein weiterer Wert eingefügt, so wird der erste Wert überschrieben
- Zugriff auf die Werte-Objekte erfolgt über die Schlüssel
- zwei wesentliche Vertreter
  - TreeMap: Einträge werden nach Schlüsseln sortiert -> Schlüssel-Klasse muss das Interface Comparable implementieren
  - HashMap: auf Basis der hashCode()-Methode der Schlüsselklasse wird eine interne Position (Bucket) berechnet, an der das Schlüssel-Werte-Paar in die Map aufgenommen wird

Wesentliche Methoden im Umgang mit Maps

- `keySet()` liefert ein Set der Schlüssel einer Map ohne Duplikate zurück
- `values()` liefert eine Collection der Werte einer Map zurück (Duplikate erlaubt)
- `put(Object k, Object v)` nimmt ein Schlüssel-Werte-Paar in die Map auf
- `get(Object k)` liefert den Wert zum Schlüssel-Objekt k zurück l containsKey(Object k) liefert true zurück, wenn zu dem Schlüssel k ein Eintrag in der Map enthalten ist
- `containsValue(Object v)` liefert true zurück, wenn zu dem Wert v ein Eintrag in der Map enthalten ist
- `remove(Object k)` löscht den Eintrag zum Schlüssel k aus der Map
- `size()` liefert die Länge der Map zurück
- `clear()` initialisiert die Map

Beispiel für eine TreeMap mit Iteratoren

```
import java.util.Set;
import java.util.Iterator;
import java.util.TreeMap;

public class DemoMap {
  public static void main(String[] args) {
    TreeMap paar = new TreeMap();
    paar.put(new Integer(130),new Hund(20, "Collie"));
    paar.put(new Integer(110),new Hund(50, "Bernhardiner"));
    paar.put(new Integer(100),new Hund(18, "Labrador"));
    paar.put(new Integer(120),new Hund(30, "Schäferhund"));
    paar.put(new Integer(130),new Hund(20, "Cocker"));

    Set schluessel = paar.keySet();
    Iterator i = schluessel.iterator();
    while (i.hasNext()) {
      Integer a = (Integer) i.next();
      Hund dog = (Hund) paar.get(a);
      System.out.println("Schlüssel: " + a + " Wert: " + dog.getRasse());
    }

    System.out.println(paar.size());
  }
}
```

Wrapper-Klassen

Umgang mit Wrapper-Klassen

- statt elementarer Datentypen werden Objekte erwartet (z.B. in Datencontainern)
- um elementare Datentypen in Objekten zu kapseln, gibt es die Wrapper-Klassen
  - stellen Methoden zur Ein- und Ausgabe sowie zur Manipulation zur Verfügung
  - stellen Methoden zur Umwandlung von Datentypen zur Verfügung
- Wrapper-Klassen existieren für folgende Datentypen
  - boolean, byte, char, double, float, int, long, short
- Auto-Boxing / Auto-Unboxing
  - Java erstellt automatisch ein Objekt der passenden Wrapper-Klasse wenn ein Objekt erwartet, aber ein einfacher Datentyp bereitgestellt wird (Auto-Boxing)
  - umgekehrt wird der Wert als einfacher Datentyp bereitgestellt, wenn ein Objekt der Wrapper-Klasse zurückgegeben wird (Auto-Unboxing)

Kapitel 10
Swing

Kapitelübersicht - Programmieren 2

8. Exception Handling
9. Collection Framework
10. **Swing**
11. Optional: Input- & Output-Stream
12. Optional: Threads

Lernziele

- Sie können den wesentlichen Unterschied zwischen AWT und Swing erläutern
- Sie können mit Swing einfache Fenster erzeugen und schließen
- Sie können unterschiedliche Layouts in Verbindung mit Panels einsetzen
- Sie können einfache Benutzerdialoge mit ausgewählten Swing-Komponenten erstellen
- Sie können validierende Textfelder erstellen
- Sie können die Interfaces Action- und ItemListener einsetzen
- Sie können eigene Menüs implementieren
- Sie können die Benutzeroberfläche mit Panels, Rahmen und Tooltips ergänzen

Kapitelübersicht - Datenstrukturen Algorithmen

1. Datenstrukturen
2. Algorithmen

Principal Collection

- KISS
- DRY
- FIRST