

# Programmieren II

Matthias Berg-Neels

[\*Download Skript\*](#)

# Kapitelübersicht - Programmieren 2

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

## Exkurse

- Unit Testing
- Innere Klassen

# Exkurs Unit Testing

*siehe "Exkurs" Skript*

# Kapitel 8

## Exception Handling

# Kapitelübersicht - Programmieren 2

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

# 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 checked und unchecked Exceptions erklären

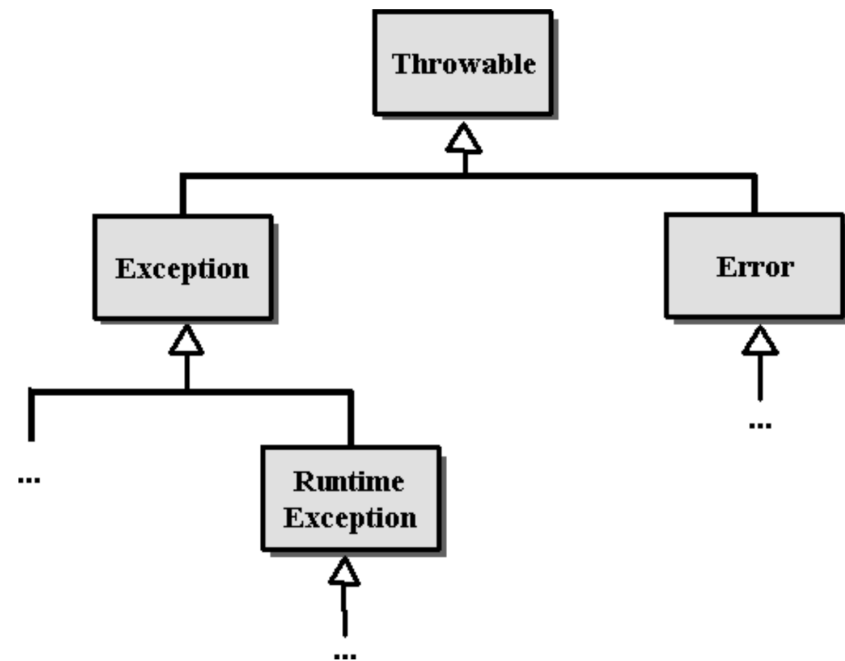
# Fehler in Java

## 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



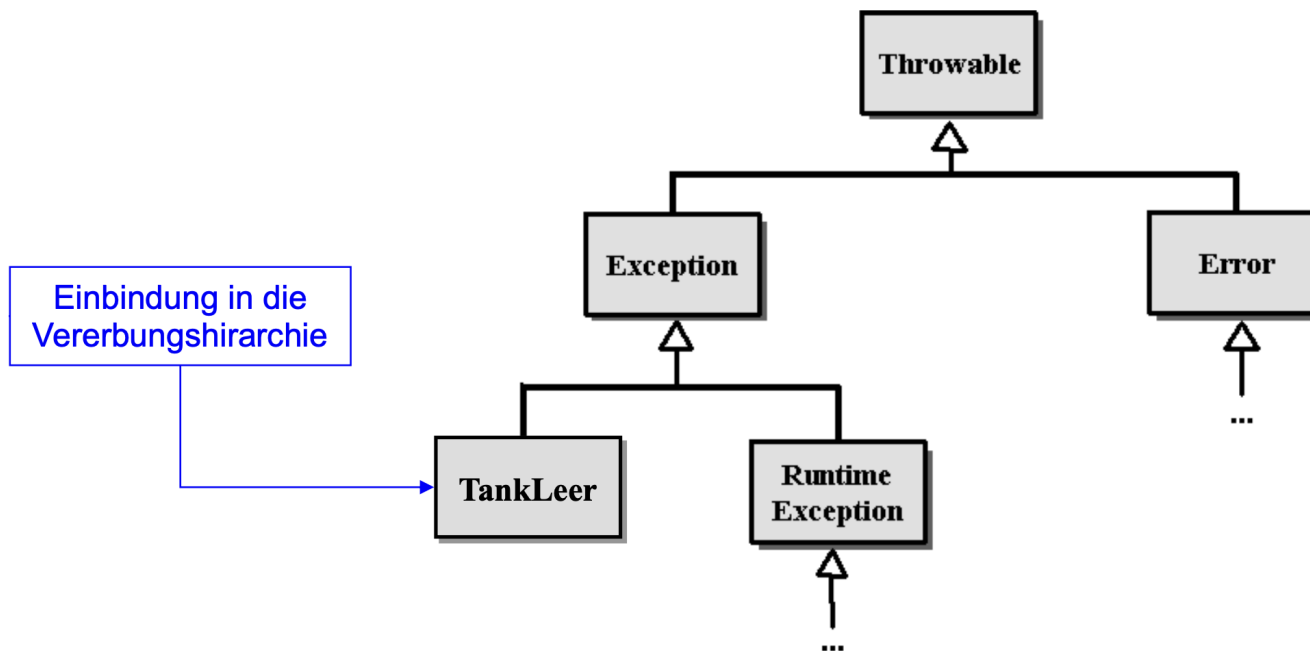
© Christian Ullenboom, Java ist auch eine Insel, 3. Auflage, S. 359

- 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 zum Abschluss des `try-catch`-Blocks ausgeführt

# try-with-Resource (Ausblick: Kapitel 11)

- vermeiden unschöner Schachtelung von try-Blöcken
  - ABER: andere Reihenfolge im Vergleich zur `finally` Ausführung
- Voraussetzung: Resource implementiert `Closable` Interface
- automatisches schließen der Ressourcen nach try-Block
  - Exceptions beim schließen werden innerhalb der Catch-Blöcke mit abgefangen

```
import java.io.*;

public class SchreibenInDatei {
    public static void main(String[] args) {
        File datei = new File(System.getProperty("user.dir") + "\\DemoLesen2.txt");

        try (FileWriter schreiber = new FileWriter(datei)) {
            datei.createNewFile();
            schreiber.write("Dies ist eine Schreibdemo.");
            schreiber.write("Es werden mehrere Zeilen geschrieben.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# 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 weitergegeben
- 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

*siehe "Exkurs" Skript*

# Kapitel 9

## Collection Framework

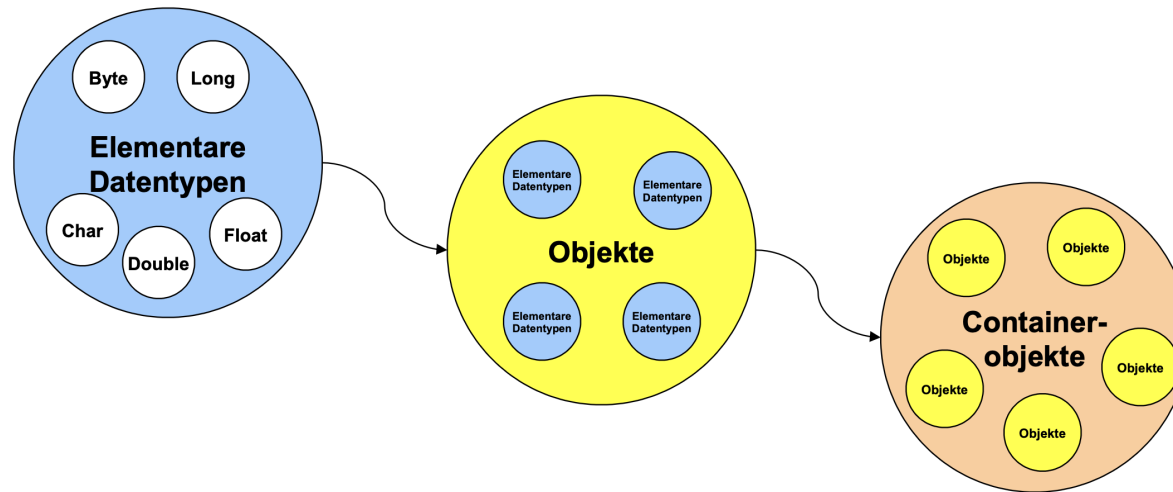
# Kapitelübersicht - Programmieren 2

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

# 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

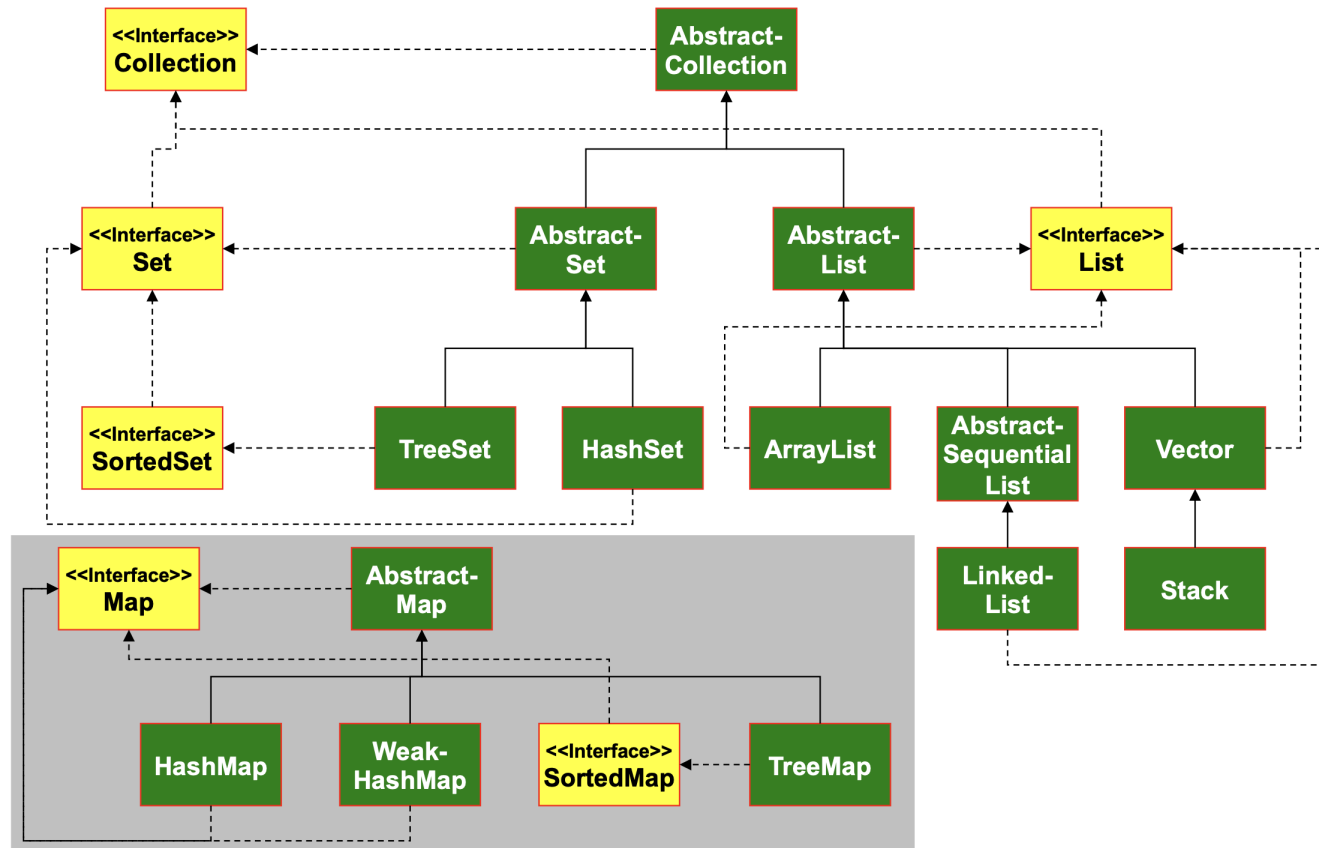
# Datenstrukturen und -container



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 wahlfrei (über Index)
- 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
- 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
- Größe der Liste wird dynamisch beim Einfügen oder Löschen von Elementen angepasst
- Duplikate sind nicht erlaubt (Vergleich über die equals-Methode)
- die Reihenfolge, in der Elemente eingefügt werden, bleibt nicht erhalten
- 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 (welche auf `SortedSet` basieren)
- soll die "natürliche Ordnung" der Objekte widerspiegeln
- Voraussetzung für Datencontainer vom Typ `SortedSet` oder `SortedMap` (sofern keine externe Sortierung über eine `Comparator` Implementierung gegeben ist)
- beinhaltet genau eine Methode: `public int compareTo(Object o)`
- Bedeutung der Rückgabewert -->  
`aufgerufenesObjekt.compareTo(übergebenesObjekt)`
  - Wert < 0: das aufgerufene Objekt ist kleiner als das übergebene Objekt
  - Wert = 0: das aufgerufene Objekt ist gleich dem übergebenen Objekt
  - Wert > 0: das aufgerufene Objekt ist größer als das übergebene Objekt
- Beispiel:

```
"a".compareTo("c"); // --> ergibt -2  
"c".compareTo("c"); // --> ergibt 0  
"c".compareTo("a"); // --> ergibt 2
```

# 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<Student> 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 Sortierreihenfolge und übersteuert die natürliche Ordnung
- `Comparator` sollten in eigener Klasse implementiert werden (auch innere Klassen oder Lamdafunktionen)
- zur Verwendung des Comparators wird ein Objekt der implementierende Klasse dem Konstruktor des Sets oder der Map übergeben
- beinhaltet genau eine Methode: `public int compare(Object o1, Object o2)`
- Bedeutung 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	<code>(field ? 0 : 1)</code>
byte, char, short, int	<code>(int) field</code>
long	<code>(int)(field&gt;&gt;&gt;32) ^ (int)(field &amp; 0xFFFFFFFF)</code>
float	<code>((x==0.0F) ? 0 : Float.floatToIntBits(field))</code>
double	<code>((x==0.0) ? 0L : Double.doubleToLongBits(field))</code> [anschliessende Behandlung wie bei long]
Referenz	<code>((field==null) ? 0 : field.hashCode())</code>



# 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?

- `compareTo()` 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 | `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. Datenstrukturen
- 13. Algorithmen

# 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

# Abgrenzung von AWT und Swing

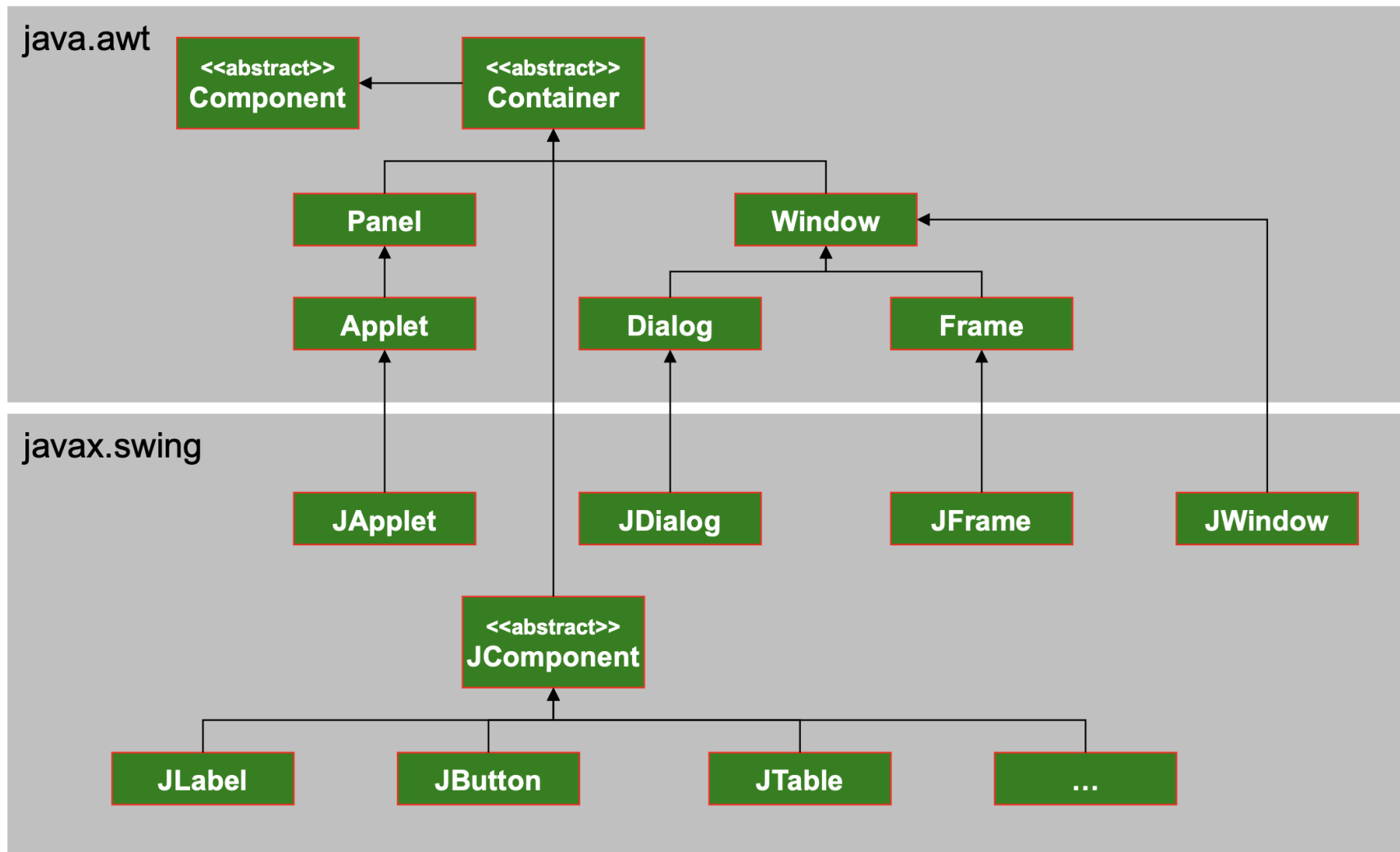
AWT (Abstract Window Toolkit) arbeitet mit „Heavyweight components“

- Verwendung von plattformspezifischen Implementierungen der AWT-Klassen (nicht in Java implementiert !)
- AWT-Komponenten besitzen einen Partner auf Betriebssystemseite (Peer), der Darstellung und Funktionalität steuert
- Vorteil: sehr schnell, da die Peer-Klassen im Code der Ausführungsplattform geschrieben sind

Swing arbeitet mit „Lightweight components“

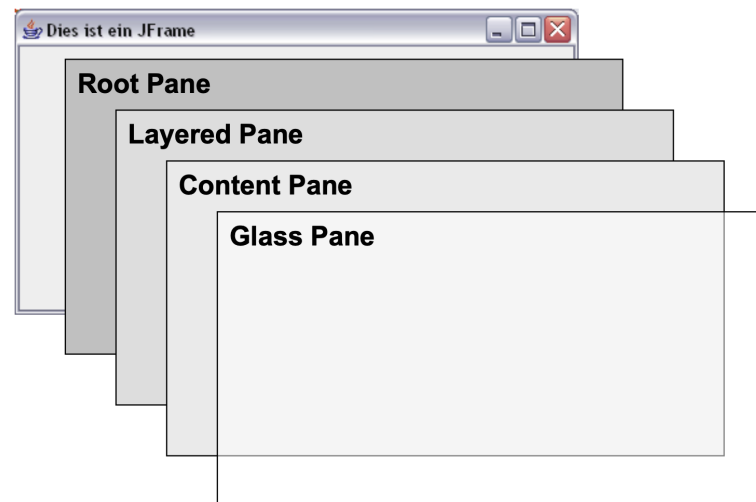
- es werden nur sehr wenige plattformspezifische GUI-Ressourcen verwendet
- lightweight components besitzen keinen Peer auf Betriebssystemseite
- Swing besitzt zahlreiche zusätzliche GUI-Komponenten
- Vorteil: „bessere“ Plattformunabhängigkeit
- Nachteil: im Vergleich zu AWT eher langsam

# Abgrenzung von AWT und Swing



# JFrame

# Aufbau eines Swing-Fensters mit JFrame



- Hauptkomponente eines JFrames ist die RootPane
- darunter folgt eine Hierarchie sogenannter Panels
- neue Komponenten werden der ContentPane zugeordnet und nicht dem JFrame
- Java Tutorial "[How to use rootPane](#)"

# Wichtige Methoden für JFrames

- überladener Konstruktor, u.a. zum Setzen des Titels
- `setDefaultCloseOperation(int i)` legt fest, was beim Schließen

des Fensters passiert

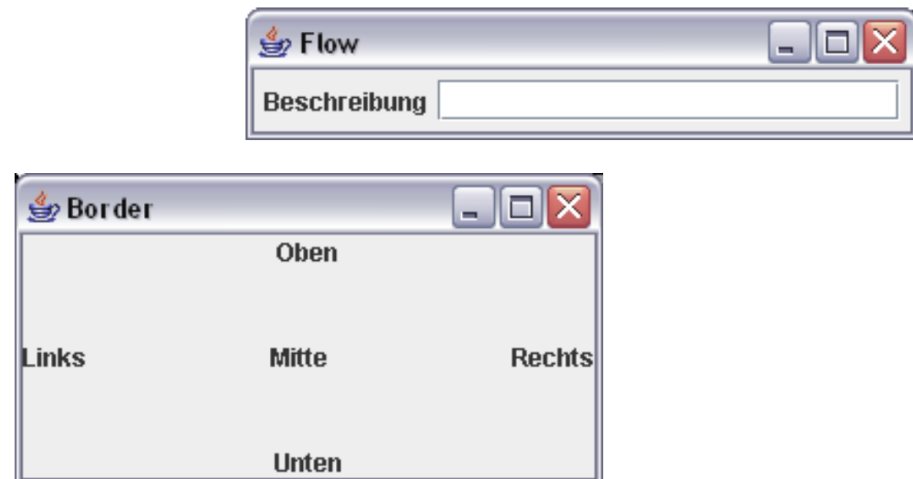
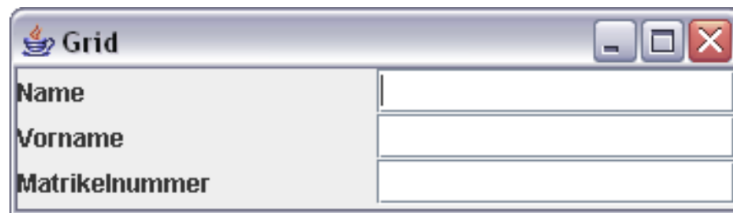
- Konstanten, die o.g. Methode übergeben werden können
  - `WindowConstants.DO_NOTHING_ON_CLOSE` löst lediglich das Close-Event aus
  - `WindowConstants.HIDE_ON_CLOSE` versteckt das Fenster
  - `WindowConstants.DISPOSE_ON_CLOSE` zerstört den Frame
  - `WindowConstants.EXIT_ON_CLOSE` beendet die Applikation
- Getter- und Setter-Methoden für die Panels eines JFrames, z.B. `getContentPane()`
- Methoden aus der Klasse `java.awt.Window`
  - `setBounds(int x, int y, int width, int height)`
  - `pack()` passt die Fenstergröße an den Content an
- `setVisible(boolean b)` aus der Klasse `java.awt.Component`



# Layoutmanager

# Layouts im Rahmen von Swing

- Anordnung der Elemente eines Containers nach bestimmten Verfahren über Layout-Manager
- wesentliche Layout-Manager
  - **FlowLayout** ordnet seine Elemente von links nach rechts
  - **BorderLayout** ermöglicht eine Anordnung in 5 verschiedenen Bereichen (NORTH, EAST, SOUTH, WEST und CENTER)
  - **GridLayout** ermöglicht die Anordnung der Komponenten in Zeilen und Spalten von links nach rechts und von oben nach unten
- mit der Methode `setLayout (LayoutManager l)` wird für ein JFrame der Layout-Manager gesetzt



# JPanel

# Der Container JPanel

- JPanel ist eine weitere Container-Form
- ordnet mehrere Elemente unter der Kontrolle eines Layoutmanagers an
- Layoutmanager und Komponenten werden direkt dem Panel zugewiesen
- bereits dem Konstruktor wird der Layoutmanager mitgegeben | über die add()-Methode werden die Komponenten dem Panel zugeordnet

# JPanel Beispiel

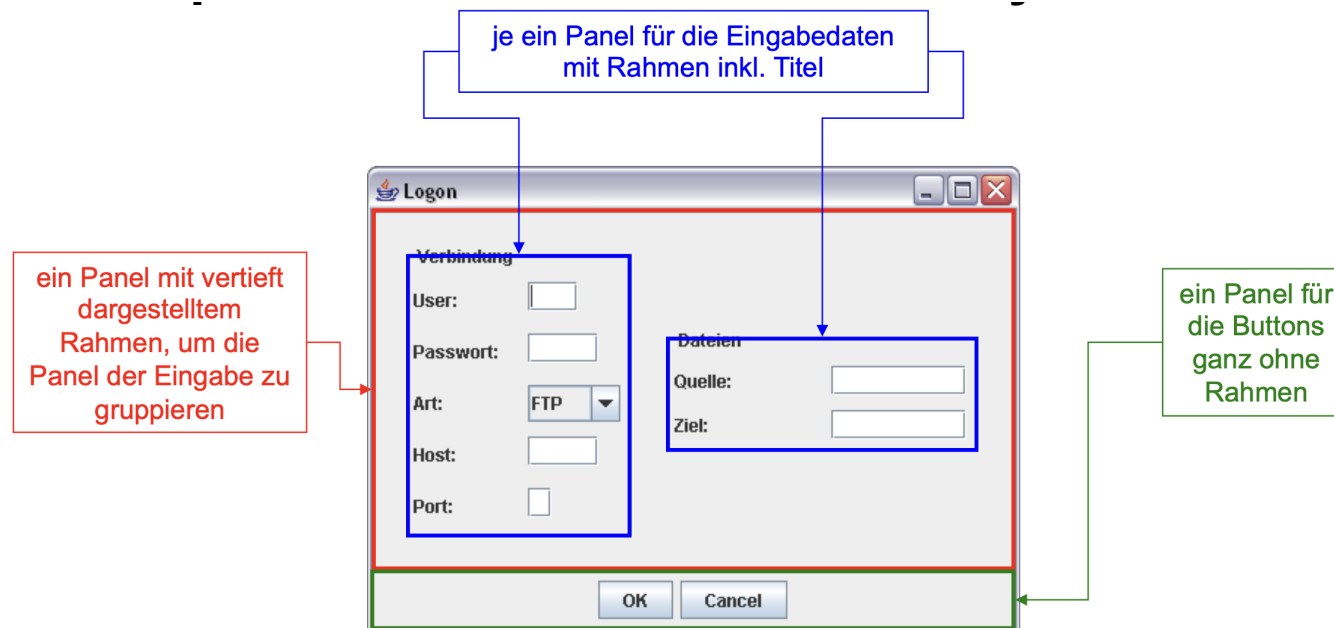
```
import java.awt.FlowLayout;
import javax.swing.*;

public class DemoFlow {
    public static void main(String[] args) {
        JFrame fenster = new JFrame("Flow");
        fenster.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        JPanel p = new JPanel(new FlowLayout(0));
        p.add(new JLabel("Beschreibung"));
        p.add(new JTextField(20));

        fenster.getContentPane().add(p);
        fenster.pack();
        fenster.setVisible(true);
    }
}
```

# Beispiel: Einsatz von JPanel und Layouts



- Hauptfenster = BorderLayout
- rot und grün umrahmtes Panel =FlowLayout
- blau umrahmte Panels = GridLayout, wobei jedes einzelne Feld auf einem eigenen Panel mit FlowLayout liegt

Rahmen mit dem **Border** Interface

# Panels mit Rahmen hervorheben

- Rahmen sind über Klassen realisiert, die das Interface **Border** implementieren
- Rahmen sollten nicht direkt über die Konstruktoren der Rahmen-Klassen sondern über die Klassenmethoden der **BorderFactory** erzeugt werden
- jeder Swing-Komponente kann mit der Methode `setBorder(Border b)` ein Rahmen zugewiesen werden
- einige Standardrahmen sind in Swing bereits implementiert



# Verschiedene Rahmen

Klasse	Rahmenart
<code>AbstractBorder</code>	eine abstrakte Klasse, die die Schnittstelle minimal implementiert
<code>BevelBorder</code>	ein 3D-Rahmen, der eingelassen sein kann
<code>CompoundBorder</code>	ein Rahmen, der andere Rahmen aufnehmen kann
<code>EmptyBorder</code>	Rahmen, dem freier Platz zugewiesen werden kann
<code>EtchedBorder</code>	noch deutlicher markierter Rahmen
<code>LineBorder</code>	Rahmen in einer einfachen Farbe in gewünschter Dicke
<code>MatteBorder</code>	Rahmen, bestehend aus Kacheln von Icons
<code>SoftBevelBorder</code>	ein 3D-Rahmen mit besonderen Ecken
<code>TitledBorder</code>	Rahmen mit String in einer gewünschten Ecke

# Beispiel: Panels mit verschiedenen Rahmen

```
import javax.swing.BorderFactory;
import javax.swing.border.BevelBorder;
import javax.swing.border.Border;

// ...

public class DemoLogonScreen {
    public DemoLogonScreen() { ...
        Border rahmen1 = BorderFactory.createEtchedBorder();
        Border rahmen2 = BorderFactory.createTitledBorder(rahmen1, "Verbindung");
        Border rahmen3 = BorderFactory.createTitledBorder(rahmen1, "Dateien");
        Border rahmen4 = BorderFactory.createTitledBorder(rahmen1, "Berechtigungen");
        Border rahmen5 = BorderFactory.createBevelBorder(BevelBorder.LOWERED);
        linkeEingabe.setBorder(rahmen2);
        rechteEingabe1.setBorder(rahmen3);
        rechteEingabe2.setBorder(rahmen4);
        mainPanel.setBorder(rahmen5);
        // ...
    }

    public static void main(String[] args) {
        DemoLogonScreen fenster = new DemoLogonScreen();
    }
}
```

# Beispiel: DemoLogonScreen

The screenshot shows a Java Swing window titled "Logon" with a standard Mac OS X-style title bar (minimize, maximize, close buttons). The window contains two main panels:

- Verbindung (Connection):** A panel on the left containing five input fields:
  - User:
  - Passwort:
  - Art:  (with a dropdown arrow)
  - Host:
  - Port:
- Dateien (Files):** A panel on the right containing two input fields:
  - Quelle:
  - Ziel:

At the bottom of the window are two buttons: "OK" and "Cancel".

# Swing UI Komponenten

# Beschriftungen und Grafikanzeige mit JLabel

- ermöglicht einfache Anzeige von Texten oder Grafiken
- zu einem Text kann zusätzlich ein Icon angezeigt werden
- bietet die Möglichkeit, HTML-Tags darzustellen
- häufiger Einsatz zur Beschriftung anderer Dialogkomponenten

```
import java.awt.GridLayout;
import javax.swing.*;

public class DemoLabelGrafik {
    public static void main(String[] args) {
        JFrame fenster = new JFrame("Bild und Label"); fenster.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE); fenster.setLa
        JLabel text = new JLabel("Hier kommt eine Grafik:");
        ImageIcon img = new ImageIcon("G:/BA/Vorlesungen/Programmierung/Demos Vorlesung/Eclipse.jpg"); JLabel bild = new JLabel(img);
        fenster.getContentPane().add(text);
        fenster.getContentPane().add(bild);
        fenster.pack();
        fenster.setVisible(true);
    }
}
```

# Unterschiedliche Arten von Textfeldern

- einfache Textfelder der Klasse `JTextField`
  - überladener Konstruktor, um das Feld mit einem String vorzubelegen und/oder die Breite anzugeben
  - Angabe der Schriftart über die Methode `setFont()`
  - Auslesen des Inhalts über die Methode `getText()`
- spezielle Felder für Passwörter der Klasse `JPasswordField`
  - Konstruktoren analog der Klasse `JTextField`
  - Auslesen des Inhalts über die Methode `getPassword()`
  - zwei boolsche Methoden `cut()` und `copy()`, die überprüfen, ob Werte mit cut (STRG+X) oder copy (STRG+C) aus dem Feld ausgelesen werden dürfen
- mehrzeilige Textfelder der Klasse `JTextArea`
  - Konstruktoren analog der Klasse `JTextField` - Unterschied: es muss neben der Breite auch die Höhe des Feldes angegeben werden
  - Auslesen und ändern der Schriftart analog der Klasse `JTextField`
  - Zeilenumbrüche werden bei `getText()` berücksichtigt

# Validierende Textfelder als spezielle Form

- realisiert durch die Klasse `JFormattedTextField`
- dem Konstruktor der Klasse wird das Format mitgegeben
- mehrere Klassen stehen für die Maskierung zur Verfügung
  - alle Objekte der Sub-Klassen der Klasse `Format` (z.B.

`SimpleDateFormat`, `DecimalFormat`, etc.)

- z.B. bei Drücken der Enter-Taste wird die Eingabe überprüft und ein mögliches `ActionEvent` ausgelöst
- Objekte der Klasse `MaskFormatter` erlauben nur bestimmte Zeichen bei der Eingabe

Platzhalter	Beschreibung
#	nur Ziffern sind erlaubt
,	Escape-Zeichen als Prefix vor einem Platzhalter
U	erlaubt nur Buchstaben, Kleinbuchstaben werden zu Großbuchstaben konvertiert
L	erlaubt nur Buchstaben, Großbuchstaben werden zu Kleinbuchstaben konvertiert
A	nur Ziffern oder Buchstaben sind erlaubt
?	nur Buchstaben sind erlaubt
*	alle Zeichen sind erlaubt
H	nur Zeichen zur Hexadezimaldarstellung sind erlaubt (0-9 und A-F)

# Drop-Down-Listen über JComboBox

- eine bestimmte Wertemenge wird zur Auswahl bereit gestellt
- dem Konstruktor der Klasse JComboBox wird die Wertemenge als ein Array von Objekten der Klasse Object übergeben
- wesentliche Methoden der Klasse JComboBox
  - `getSelectedItem()` liefert den Wert des ausgewählten Elements zurück (entspricht der Methode `getText()` bei `JTextField`)
  - `setSelectedItem(Object o)` belegt das Feld mit dem Wert `o` vor, sofern dieser in dem Array der Wertemenge vorhanden ist
  - `setEditable(boolean b)` bestimmt, ob auch Werte außerhalb der Wertemenge erlaubt sind
    - `b = true` -> freie Eingabe erlaubt
    - `b = false` -> freie Eingabe nicht erlaubt



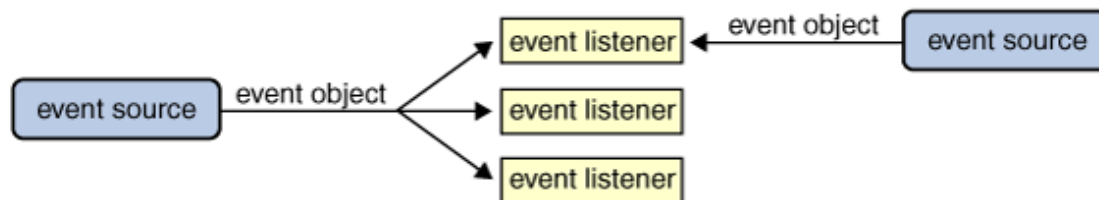
# Weitere Klassen aus dem Package Swing

- **JTable**
  - dient der Erstellung zweidimensionaler Tabellen
- **JTree**
  - ermöglicht die Darstellung von Bäumen ähnlich dem Windows Explorer bestehend aus Knoten und Blättern
- **JToolBar**
  - dient der Erstellung von Symbolleisten analog den Microsoft Office-Produkten
- **JColorChooser**
  - dient der Erstellung eines Auswahldialogs zur Farbeinstellung
- **JFileChooser**
  - dient der Erstellung eines Dialogs zur Auswahl einer Datei im FileSystem
- ...

# EventListener

# Das Event Konzept

- UI Komponenten erzeugen Events z.B. klick auf einen Button ([ActionEvent](#)), ändern einer Auswahl ([ItemEvent](#)), wechseln eines Fensters ([FocusEvent](#)) ...
- Jeder Event-Typ hat ein Listener Interface um auf das Event reagieren zu können
  - [ActionEvent](#) --> [ActionListener](#)
  - [ItemEvent](#) --> [ItemListener](#)
  - [FocusEvent](#) --> [FocusListener](#)
  - ...
- Komponenten, welche Events erzeugen, können (mehrere) Implementierungen des jeweiligen Interfaces registrieren (z.b. `addActionListener(ActionListener a)`)
- Tritt ein Event auf, wird die jeweilige Methode der registrierten Interface-Implementierung aufgerufen



# Die Aufgaben des **ItemListener**

- der ItemListener ist als Interface implementiert
- das Interface gibt die abstrakte Methode `itemStateChanged(ItemEvent e)` vor
- das Interface wird von Objekten implementiert, die an einem Auswahlereignis interessiert sind
- Auswahlereignisse können von Objekten folgender Klassen ausgelöst werden: `JComboBox`, `JCheckBox`, `JList` oder `JCheckBoxMenuItem`
- die Zuordnung zu einem ItemListener erfolgt über die jeweiligen Objekt-Methoden `addItemListener()` oder `removeItemListener()`
- wird ein Eintrag bei o.g. Objekten ausgewählt, wird implizit die Methode `itemStateChanged(ItemEvent e)` bei allen bei dem Objekt registrierten ItemListenern ausgeführt
- Beispiel: beim Setzen des Hakens wird ein zusätzliches Feld eingeblendet

# Beispiel: JComboBox mit ItemListener

```
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

// ...

public class DemoJComboBox {
    // ...
    public DemoJComboBox() {
        // ...
        ItemListener zuhoerer = new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                JComboBox auswahl = (JComboBox)e.getSource();
                if(auswahl.getSelectedItem().equals("sonstiges")) {
                    sonstLabel.setVisible(true);
                    sonst.setVisible(true);
                } else {
                    sonstLabel.setVisible(false);
                    sonst.setVisible(false);
                }
            }
        };

        Object[] werte = {"DVD", "VCD", "VHS", "SVCD", "sonstiges"};

        JComboBox medium = new JComboBox(werte);
        medium.addItemListener(zuhoerer);
        // ...
    }

    public static void main(String[] args) {
        DemoJComboBox fenster = new DemoJComboBox();
    }
}
```

# Interaktion über Drucktasten mit **JButton**

- überladener Konstruktor, der es ermöglicht Text und oder Grafik in Form eines Icon auf dem Button zu positionieren
- mit der Methode `setText(String s)` kann der Text nachträglich verändert werden
- wichtigste Methoden `addActionListener()` und `removeActionListener()`
- der `ActionListener` ist der Beobachter des Knopfes
- ohne `ActionListener` kann dem Button keine Funktionalität zugewiesen werden
- sobald der Button gedrückt wird, wird ein `ActionEvent` ausgelöst, welches vom Beobachter abgefangen und ausgewertet wird

# Die Aufgaben des ActionListener

- der ActionListener ist als Interface implementiert
- das Interface gibt die abstrakte Methode `actionPerformed(ActionEvent e)` vor
- diese Methode wird implizit ausgeführt, sobald ein „abgehörtes“ Objekt ein `ActionEvent` auslöst
- die Klasse `ActionEvent` besteht aus drei Methoden
  - `getActionCommand()` liefert den String, der mit der Aktion verbunden ist (bei `JButton` die Beschriftung des Buttons)
  - `getModifiers()` liefert einen Integer-Wert zurück, welche Funktionstaste bei dem Ereignis gedrückt wurde (Shift, Alt, etc.)
  - `paramString()` liefert einen Erkennungs-String, der mit „ACTION\_PERFORMED“ oder „unknown type“ beginnt

# Beispiel für einen JButton mit ActionListener

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
// ...

public class DemoButton { public DemoButton() {
    public DemoButton() {
        // ...
        ActionListener zuhoerer = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String ereignis = e.getActionCommand();
                if (ereignis.equals("OK")) {
                    System.out.println("Es wurde OK gedrückt.");
                } else {
                    System.exit(0);
                }
            }
        };

        JButton ok = new JButton("OK");
        ok.addActionListener(zuhoerer);
        JButton exit = new JButton("Exit");
        exit.addActionListener(zuhoerer);
        //...
    }

    public static void main(String[] args) {
        DemoButton fenster = new DemoButton(); }
    }
}
```



# Kontrollfelder mit JCheckBox

- Kontrollfelder kennen zwei Zustände: selektiert (`true`) und nicht selektiert (`false`)
- überladener Konstruktor, der es ermöglicht Text, Initialwert (`true` oder `false`) und Icon mitzugeben
- Kontrollfelder werden normalerweise als Kästchen mit einem Häkchen für den selektierten Zustand dargestellt
- der Zustand kann über die Methode `setSelected(boolean b)` geändert werden
- der Zustand kann allerdings nicht direkt über eine Getter-Methode ausgelesen werden
- bei der Änderung des Zustands durch den Anwender wird ein `ItemEvent` ausgelöst und an alle registrierten `ItemListener` weitergeleitet
- im `ItemListener` kann der Zustand des Kontrollfeldes ausgewertet und weiter verarbeitet werden

# Beispiel: JCheckBox mit ItemListener

```
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener; ...

public class DemoJCheckBox {
    // ...

    private ItemListener hoerer1 = new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
            if (e.getStateChange() == ItemEvent.SELECTED) {
                ueber.setText("Datei wird überschrieben");
            } else {
                ueber.setText("Datei wird nicht überschrieben");
            }
        }
    };
    // ...

    public DemoJCheckBox() {
        // ...
        JCheckBox ueber = new JCheckBox("Datei wird nicht überschrieben", false);
        ueber.addItemListener(hoerer1);
        // ...
    }

    public static void main(String[] args) {
        DemoJCheckBox fenster = new DemoJCheckBox();
    }
}
```

# Optionsfelder mit JRadioButton & ButtonGroup

- Optionsfelder bieten mehrere Auswahlmöglichkeiten an, wobei nur eine Option ausgewählt werden kann
- dazu werden Optionsfelder in einem Objekt der Klasse `ButtonGroup` zu einer Optionsfeldgruppe zusammengefasst
  - mit der Objektmethode `add(AbstractButton b)` der Klasse `ButtonGroup` wird ein Optionsfeld der Gruppe hinzugefügt
  - mit der Objektmethode `remove(AbstractButton b)` der Klasse `ButtonGroup` wird ein Optionsfeld aus der Gruppe entfernt
- überladener Konstruktor der Klasse `JRadioButton` analog der Klasse `JCheckBox`
- Optionsfelder werden normalerweise als Kreis mit einem schwarzen Punkt für den selektierten Zustand dargestellt
- bei der Änderung des Zustands eines Optionsfeldes wird ein `ActionEvent` ausgelöst und an alle registrierten `ActionListener` weitergeleitet
- im `ActionListener` kann die Auswertung der Optionsfelder erfolgen

# Beispiel: JRadioButton mit ActionListener

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
//...

public class DemoRadioButton {

    private ActionListener hoerer2 = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if (opt1 == e.getSource()) {
                System.out.println("Datei kann nur gelesen werden");
            } else if (opt2 == e.getSource()) {
                System.out.println("Datei kann nur geschrieben werden");
            } else if (opt3 == e.getSource()) {
                System.out.println("Datei kann gelesen und geschrieben werden");
            }
        }
    };
    // ...

    public DemoRadioButton() {
        // ...
        opt1 = new JRadioButton("Nur Lesen",true);
        opt1.addActionListener(hoerer2);
        opt2 = new JRadioButton("Nur Schreiben",false);
        opt2.addActionListener(hoerer2);
        optGroup = new ButtonGroup();
        optGroup.add(opt1);
        optGroup.add(opt2);
        optGroup.add(opt3);
        // ...
    }

    public static void main(String[] args) {
        DemoRadioButton fenster = new DemoRadioButton();
    }
}
```

# Weitere Komponenten

# Erstellen von Menüs mit Swing-Komponenten

- **JMenuBar** ist der Container für die einzelnen Menüs
  - mit der `add(JMenu m)` Methode wird dem Container ein Menü hinzugefügt
- Objekte der Klasse **JMenu** stellen die einzelnen Menüs dar und sind Container für konkrete Menüeinträge
  - mit der `add(JMenuItem i)` Methode wird einem Menü ein konkreter Menüeintrag zugeordnet
- Objekte der Klasse **JMenuItem** repräsentieren Menüeinträge
- mit der Methode `setJMenuBar(JMenuBar m)` wird einem Fenster eine Menüleiste zugeordnet
- um auf die Auswahl eines Menüeintrags zu reagieren, müssen die Menüeinträge einem `ActionListener` zugeordnet werden

# Beispiel: einfaches Menü mit ActionListener

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class DemoJMenuBar {
    private ActionListener hoerer = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String ereignis = e.getActionCommand();
            if (ereignis.equals("Beenden")){
                System.exit(0);
            } else {
                System.out.println(ereignis);
            }
        }
    };

    public DemoJMenuBar() { ...
        JMenuBar menue = new JMenuBar();
        // ...
        JMenu bea = new JMenu("Bearbeiten");
        JMenuItem aus = new JMenuItem("Ausschneiden"); ...
        aus.addActionListener(hoerer); bea.add(aus);
        // ...
        menue.add(bea);
        // ...
        fenster.setJMenuBar(menue);
    }
    // ...
}
```

# Tooltips

- Tooltips sind kleinere Hilfetexte, die beim längeren Verweilen auf einem GUI-Objekt in einem kleinen PopUp-Fenster angezeigt werden
- ToolTips werden nicht direkt über den Konstruktor der Klasse `JToolTip` erzeugt, sondern über die Methode `setToolTipText(String s)` des GUI-Objektes
  - der String `s` kann als einfacher Text übergeben werden
  - der String `s` kann im HTML-Format übergeben werden

```
import javax.swing.JButton;

public class DemoToolTip {
    public DemoToolTip() {
        //...
        JButton ok = new JButton("OK");
        ok.addActionListener(zuhoerer);
        ok.setToolTipText("Führt die Funktion aus");
        // ...
    }

    public static void main(String[] args) {
        DemoToolTip fenster = new DemoToolTip();
    }
}
```



# Kapitel 11

## Input- & Output-Streams (!)

# Kapitelübersicht - Programmieren 2

- 8. Exception Handling
- 9. Collection Framework
- 10. Swing
- 11. **Optional: Input- & Output-Streams**
- 12. Datenstrukturen
- 13. Algorithmen

# Lernziele

- Sie können aus Java heraus auf das File-System zugreifen
- Sie können Verzeichnisse und Dateien anlegen, umbenennen und löschen
- Sie können Ein- und Ausgaben auf der Konsole vornehmen
- Sie können sowohl schreibend als auch lesend auf Textdateien zugreifen
- Sie können Dateien kopieren
- Sie können das Konzept der Serialisierung in Java anwenden
- Sie können Properties-Dateien anlegen, mit Werten füllen und wieder auslesen
- Sie können Anwendungen mit mehrsprachigen Texten implementieren

# Umgang mit dem Dateisystem (!)

# Windows Dateisystem

The image shows a Windows Explorer window titled 'BA' displaying the file system structure of a drive. The address bar shows 'G:\BA'. The left pane shows a tree view of folders, including 'Studienarbeiten' and 'Vorlesungen'. The right pane shows a list of files and folders with columns for Name, Größe, Typ, and Geändert. A red box labeled 'Laufwerke' points to the drive letter 'G:' in the address bar. A blue box labeled 'Verzeichnisse' points to the 'Studienarbeiten' folder. A green box labeled 'Dateien' points to the 'Studienplan\_2003\_Kurz.doc' file.

**Laufwerke**

**Verzeichnisse**

**Dateien**

**Eingabeaufforderung**

```
G:\BA>dir
Datenträger in Laufwerk G: ist Michael
Volumeseriennummer: B830-939B

Verzeichnis von G:\BA

08.12.2005  19:23    <DIR>          .
08.12.2005  19:23    <DIR>          ..
16.12.2005  06:19    <DIR>          Studienarbeiten
24.07.2003  08:50      244.736 Studienplan_2003_Kurz.doc
17.02.2005  09:49      226.688 Studienplan_2003_Kurz.pdf
24.07.2003  08:50      329.728 Studienplan_2003_Lang.doc
17.02.2005  09:52      339.753 Studienplan_2003_Lang.pdf
26.01.2006  17:21    <DIR>          Vorlesungen
               4 Datei(en)      1.140.905 Bytes
               4 Verzeichnis(se), 60.516.405.248 Bytes frei

G:\BA>
```

# Die Klasse `File`

Objekte der Klasse repräsentieren

- Laufwerke
- Verzeichnisse
- Dateien

Wichtige Methoden der Klasse `File`

- zum Erzeugen, Umbenennen und Löschen von
  - Verzeichnissen
  - Dateien
- zum Beschaffen von Informationen über die Objekte
- Klassenmethoden zur Auflistung von Inhalten

# Anzeigen der Laufwerke

```
import java.io.File;

public class AusgabeVerzeichnis {
    public static void main(String[] args) {

        File[] laufwerke = File.listRoots();

        for (int i = 0; i < laufwerke.length; i++) {
            System.out.println(laufwerke[i].getPath()
                + (laufwerke[i].exists() ? " ist aktiviert" : " ist deaktiviert"));
        }
    }
}
```

# Informationen über Verzeichnisse

```
import java.io.File;

public class VerzeichnisEigenschaften {
    public static void main(String[] args) {
        File verzeichnis = new File("G:/BA/Vorlesungen/Programmierung/Skript");

        if (verzeichnis.exists() && verzeichnis.isDirectory()) {
            System.out.println("Vorgänger:\t" + verzeichnis.getParent());
            System.out.println("Pfad:\t\t" + verzeichnis.getPath());
            System.out.println("Name:\t\t" + verzeichnis.getName());
            File[] liste = verzeichnis.listFiles();
        } else {
            System.out.println(" Das Verzeichnis "
                + verzeichnis.getPath() + " existiert nicht.");
        }
    }
}
```

- `isDirectory()` überprüft, ob das Objekt ein Verzeichnis ist
- `getParent()` liefert Pfad des Vorgängers als String zurück
- `getPath()` liefert den Pfadnamen als String zurück
- `getName()` liefert den Namen als String zurück
- `listFiles()` erzeugt ein Array vom Typ `File` mit dem Verzeichnisinhalt



# Informationen über Dateien

```
import java.io.File;

public class DateiEigenschaften {
    public static void main(String[] args) throws Exception {

        File datei = new File("G:/BA/Vorlesungen/Programmierung/Skript/Programmierung 1.ppt");

        if (datei.exists() && datei.isFile()) {
            System.out.println("Name der Datei:\t\t" + datei.getName()
                + "\nSpeicherort der Datei:\t" + datei.getPath()
                + "\nPfad der Datei:\t\t" + datei.getParent()
                + "\nGrösse der Datei:\t" + datei.length() + " Byte"
                + "\nBerechtigung (r/w):\t" + datei.canRead() + " " + datei.canWrite()
                + "\nZuletzt geändert:\t" + datei.lastModified());
        } else {
            System.out.println(" Die Datei " + datei.getName() + " existiert nicht.");
        }
    }
}
```

- `isFile()` überprüft, ob das Objekt eine Datei ist
- `length()` gibt die Länge der Datei in Byte an
- `canRead()` überprüft die Leseberechtigung
- `canWrite()` überprüft die Schreibberechtigung
- `lastModified()` gibt den Zeitpunkt der letzten Änderung an

# Umgang mit Verzeichnissen

```
import java.io.File;

public class Verzeichnis {
    public static void main(String[] args) {

        File verzeichnis = new File(System.getProperty("user.dir"));
        File neuerOrdner = new File(verzeichnis.getPath() + "/demoPfad/");
        File neuerOrdner2 = new File(verzeichnis.getPath() + "/demoPfad2/");

        if (!neuerOrdner.exists()) {
            neuerOrdner.mkdir();
            System.out.println("Der Pfad wurde angelegt.");
        }

        if (!neuerOrdner2.exists()) {
            neuerOrdner.renameTo(neuerOrdner2);
            System.out.println("Der Pfad wurde umbenannt.");
        }

        if (neuerOrdner2.exists()) {
            neuerOrdner2.delete(); // Setzt voraus, dass der Ordner leer ist
            System.out.println("Der Pfad wurde gelöscht.");
        }
    }
}
```

# Umgang mit Dateien

```
import java.io.File;
import java.io.IOException;

public class Dateien {
    public static void main(String[] args) {
        File verzeichnis = new File(System.getProperty("user.dir"));
        File neueDatei = new File(verzeichnis.getParent() + "/MeineDatei.txt");
        File neueDatei2 = new File(verzeichnis.getParent() + "/MeineDatei2.txt");

        try {
            if (!neueDatei.exists()) {
                neueDatei.createNewFile();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        if (!neueDatei2.exists()) {
            neueDatei.renameTo(neueDatei2);
        }

        if (neueDatei2.exists()) {
            neueDatei2.delete();
        }
    }
}
```

# Input- & Output-Streams (!)

# Streams (`java.io`) VS Streams (`java.util.stream`)

Stream - Grundkonzept

- übertragen von Elementen (Daten) zwischen einer Quelle und einem Ziel

Stream (`java.io`)

- übertragen von Daten zu bzw. von externen Ressourcen
- I/O --> Input / Output
- Beispiele
  - lesen aus Dateien
  - Ausgabe auf der Konsole
  - senden von Daten über das Netzwerk

Stream (`java.util.stream`)

- einfache (besser lesbare) Modifikation von Daten(-Strömen)
  - Ersatz (funktionales Paradigma) für sequentielle Abarbeitung wie z.B. Schleifen
- Erweiterung des Collection Framework
- Quelle von Interface Collection, Ziel von Interface Collection, einfache Datentypen, etc (abhängig von Funktion)

# Ein- und Ausgabeströme in Java

## Eingabestrom

- über Tastatur in Verbindung mit der Konsole
- aus existierenden Dateien
- wichtige Klassen
  - Byte oder Byte-Arrays (InputStream)
  - Zeichen oder Zeichen-Arrays (Reader)

## Ausgabestrom

- auf die Konsole
- in existierende oder neue Dateien
- wichtige Klassen
  - Byte oder Byte-Arrays (OutputStream)
  - Zeichen oder Zeichen-Arrays (Writer)

# Übersicht über wichtige Eingabeklassen

Byte-Stream-Klasse für die Eingabe	Zeichen-Stream- Klasse für die Eingabe	Beschreibung
InputStream	Reader	Abstrakte Klasse für Zeicheneingabe und Byte-Arrays
BufferedInputStream	BufferedReader	Puffert die Eingabe
LineNumberInputStream	LineNumberReader	Merkt sich Zeilennummern beim Lesen
ByteArrayInputStream	CharArrayReader	Liest Zeichen-Arrays oder Byte-Arrays
(keine Entsprechung)	InputStreamReader	Wandelt Byte-Stream in Zeichen-Stream um, Bindeglied zwischen Byte und Zeichen
FileInputStream	FileReader	Liest aus einer Datei

# Übersicht über wichtige Ausgabeklassen

Byte-Stream-Klasse für die Ausgabe	Zeichen-Stream- Klasse für die Ausgabe	Beschreibung
OutputStream	Writer	Abstrakte Klasse für Zeichenausgabe oder Byte-Ausgabe
BufferedOutputStream	BufferedWriter	Puffert die Eingabe
ByteArrayOutputStream	LineNumberReader	Ausgabe des Puffers, nutzt passendes Zeilenendezeichen
ByteArrayInputStream	CharArrayWriter	Schreibt Arrays
(keine Entsprechung)	OutputStreamWriter	Übersetzt Zeichen-Stream in Byte-Stream
FileOutputStream	FileWriter	Schreibt in eine Datei



# Ein- und Ausgabe auf der Konsole

Vordefinierte In- und Outputstreams in der Klasse `System`

Besondere Stream-Klassen für Standardgeräte

- `System.in` für die Tastatur
  - Vom Typ `BufferedInputStream`
  - Vorsicht: `Checked Exception`
- `System.out` für den Monitor

Werden automatisch beim Laden von Klassen erzeugt

Besonderer Output-Stream `System.err`

# Ausgaben auf die Konsole mit System.out

2 Möglichkeiten der Ausgabe

- `System.out.print()`; ohne Zeilenumbruch
- `System.out.println()`; mit Zeilenumbruch

`print()` und `println()` sind überladen für

- elementare Datentypen
- Argumente der Klasse `String`
- Argumente der Klasse `Object`

Konvertierung der Übergabeparameter

- Parameter werden in einen `String` konvertiert
- Konvertierung durch impliziten Aufruf der Methode `toString()`

# Eingaben über die Konsole mit `System.in`

```
import java.io.IOException;

public class EingabeTastatur {
    public static void main(String[] args) {
        byte[] eingabe = new byte[255];

        System.out.print("Geben Sie einen Text ein:");

        try {
            System.in.read(eingabe, 0, 255);
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println(eingabe);
        System.out.println(new String(eingabe));
    }
}
```

# Eingaben über die Konsole mit `System.in`

```
import java.io.*;

public class EingabeTastaturString {
    public static void main(String[] args) {

        InputStreamReader strRead = new InputStreamReader(System.in);
        BufferedReader bufString = new BufferedReader(strRead);

        String eingabe = "";

        System.out.println("Geben Sie Ihren Text ein: ");

        try {
            eingabe = bufString.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println(new String(eingabe));
    }
}
```

# Streams und Dateien (!)

# Fortgeschrittener Umgang mit Dateien

## Lesen aus Dateien

- aus einfachen Textdateien
  - Öffnen der Datei über Objekte der Klasse `FileReader`
  - Pufferung der gelesenen Daten im `BufferedReader`
- aus beliebigen Dateien
  - Öffnen der Datei über Objekte der Klasse `FileInputStream`
  - mögliche Pufferung in Objekten der Klasse `BufferedInputStream`

## Schreiben in Dateien

- in einfache Textdateien
  - Schreiben von Strings über Objekte der Klasse `FileWriter`
- in beliebige Dateien
- Schreiben von Daten über Objekte der Klasse `FileOutputStream`

# Lesen aus Textdateien

```
import java.io.*;

public class LesenAusDatei {
    public static void main(String[] args) {
        File datei = new File(System.getProperty("user.dir") + "\\DemoLesen.txt");

        String text = new String();

        try {
            FileReader leser = new FileReader(datei);
            BufferedReader lesePuffer = new BufferedReader(leser);

            String line;

            while ((line = bufferedRenameFileReader.readLine()) != null) {
                System.out.println(line);
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Schreiben in Textdatei

```
import java.io.*;

public class SchreibenInDatei {
    public static void main(String[] args) {
        File datei = new File(System.getProperty("user.dir") + "\\DemoLesen2.txt");
        FileWriter schreiber = null;

        try {
            schreiber = new FileWriter(datei);
            datei.createNewFile();
            schreiber.write("Dies ist eine Schreibdemo.");
            schreiber.write("Es werden mehrere Zeilen geschrieben.");
        } catch (IOException e) { e.printStackTrace(); }
        finally {
            try {
                schreiber.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```



# Rückblick - Exception Handling: try-with-Resource

- vermeiden unschöner Schachtelung von try-Blöcken
- Voraussetzung: Resource implementiert `Closable`
- automatisches schließen der Ressourcen nach try-Block, Exceptions beim schließen werden innerhalb der Catch-Blöcke mit abgefangen

```
import java.io.*;

public class SchreibenInDatei {
    public static void main(String[] args) {
        File datei = new File(System.getProperty("user.dir") + "\\DemoLesen2.txt");

        try (FileWriter schreiber = new FileWriter(datei)) {
            datei.createNewFile();
            schreiber.write("Dies ist eine Schreibdemo.");
            schreiber.write("Es werden mehrere Zeilen geschrieben.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Einfache Möglichkeit zum Kopieren von Dateien

```
import java.io.*;

public class DateiKopieren {
    public static void main(String[] args) {
        File quelle = new File(System.getProperty("user.dir") + "/Eclipse.jpg");
        File ziel = new File(System.getProperty("user.dir") + "/Eclipse2.jpg");

        byte[] puffer = new byte[(int)quelle.length()];

        try (FileInputStream leser = new FileInputStream(quelle);
            FileOutputStream schreiber = new FileOutputStream(ziel)) {

            int bytesRead = leser.read(puffer);
            schreiber.write(puffer, 0, bytesRead);

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Kopieren von Dateien über Puffer

```
import java.io.*;

public class DateiKopierenMitPuffer2 {
    public static void main(String[] args) {
        final int BUF_SIZE = 1;
        File quelle = new File(System.getProperty("user.dir") + "/Eclipse.jpg");
        File ziel = new File(System.getProperty("user.dir") + "/Eclipse2.jpg");

        int i = 0;
        int puffer = 0;
        byte[] buffer = new byte[BUF_SIZE];

        try (FileInputStream leser = new FileInputStream(quelle);
            FileOutputStream schreiber = new FileOutputStream(ziel)) {

            while (true){
                puffer = leser.read(buffer, i, BUF_SIZE);
                if (puffer == -1){
                    break;
                }
                schreiber.write(buffer, i, BUF_SIZE);
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Serialisierung von Objekten (!)

# Übertragung von Objekten

- Umwandlung von Objekten in Byte-Strom (serialisieren)
- Umwandlung von Byte-Strom in Objekte (deserialisieren)

## Umsetzung in Java

- "implementieren" von `Serializable`
- automatische / manuelle Versionierung über `static final long serialVersionUID`
  - automatische Versionierung in Java über Änderungen an der Klassen (HashCode)
- Umsetzung in Java über das Reflection-Framework
- Möglichkeiten zur Klassen-spezifischen Implementierung
  - Modifier `transient` zum ausschließen von Feldern
  - Methoden im Objekt-Lebenszyklus `readResolve()`, `writeObject(ObjectOutputStream s)`, `readObject(ObjectInputStream s)`

# Serialisierungs-Beispiel: Person

```
import java.io.Serializable;

public class Person implements Serializable {

    static final long serialVersionUID = 1L;

    private String name;
    private String familyName;
    private transient String fullName;

    public Person(String name, String familyName){
        this.setName(name);
        this.setFamilyName(familyName);
        this.setFullName();
    }

    private Object readResolve(){
        this.setFullName();
        return this;
    }

    private void setFullName() {
        this.fullName = this.getFamilyName() + "; " + this.getName();
    }

    // ... Getter / Setter / to String
}
```

# ObjectOutputStream: Speichern von Personen Objekten

```
import java.io.*;

public class PersonSaveExample {

    public static void main(String[] args) {

        File personPersistentFile = new File(System.getProperty("user.dir") + File.separator + "Person.dat");

        try(FileOutputStream personFileOutputStream = new FileOutputStream(personPersistentFile);
            ObjectOutputStream personObjectOutputStream = new ObjectOutputStream(personFileOutputStream)){

            personObjectOutputStream.writeObject(new Person("Klaus", "Müller"));
            personObjectOutputStream.writeObject(new Person("Gabi", "Mayer"));
            personObjectOutputStream.writeObject(new Person("Hans", "Franz"));

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

# ObjectInputStream: Lesen von Personen-Objekten

```
import java.io.*;

public class PersonReadExample {

    public static void main(String[] args) {

        File personPersistentFile = new File(System.getProperty("user.dir") + File.separator + "Person.dat");

        try(FileInputStream personFileInputStream = new FileInputStream(personPersistentFile);
            ObjectInputStream personObjectInputStream = new ObjectInputStream(personFileInputStream)){

            while(true){
                try{
                    System.out.println(personObjectInputStream.readObject());
                }catch (EOFException e){
                    break;
                }
            }

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }
}
```



# Properties (!)

# Texte in Java-Properties-Dateien auslagern (!)

## Vorteile der Properties

- Ziel: Auslagerung von Texten in eigener Datei (.properties)
- Ablage von Schlüssel-Wertepaaren (Alias & Wert) als Strings
- ab Java 1.5 können die Properties auch im XML-Format abgelegt werden
- Texte können ohne Kompilierung des Bytecodes verändert werden

## Umsetzung in Java

- Nutzung der Klasse Properties und des FileInputStream- bzw. FileOutputStreams
- Laden, Setzen und Speichern von Properties möglich
- dynamische Texte mit variablen Parametern möglich

# Texte in Java-Properties-Dateien speichern (!)

```
import java.io.*;
import java.util.*;

public class PropertiesSpeichernDemo {
    public static void main(String[] args) {

        File propDateiName = new File(System.getProperty("user.dir") + "\\Demo2.properties");

        try (FileOutputStream propDatei = new FileOutputStream(propDateiName)) {
            Properties prop = new Properties();
            prop.setProperty("Name", "Michael Lang");
            prop.setProperty("Language", "Deutsch");
            prop.store(propDatei, "Dies ist der Kommentar");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ergebnis (in Demo2.properties Datei):

```
#Dies ist der Kommentar
#Sun Mar 19 17:29:02 CET 2006
Name=Michael Lang
Language=Deutsch
```

# Texte aus Java-Properties-Dateien lesen (!)

```
import java.io.*;
import java.util.*;

public class PropertiesLadenDemo {
    public static void main(String[] args) {
        File propDateiName = new File(System.getProperty("user.dir") + "\\Demo2.properties");

        try (FileInputStream propDatei = new FileInputStream(propDateiName)){

            Properties prop = new Properties();
            prop.load(propDatei);
            prop.list(System.out);

            System.out.println("\nHallo " + prop.getProperty("Name"));
            System.out.println("Sie bekommen die Texte in " + prop.getProperty("Language") + " angezeigt.");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ergebnis (Ausgabe auf Konsole):

```
-- listing properties --
Language=Deutsch
Name=Michael Lang
Hallo Michael Lang
Sie bekommen die Texte in Deutsch angezeigt.
```

# Dynamische Texte mit variablen Parametern (!)

Demo.properties Datei:

```
#Dies ist der Kommentar  
dyna=Text mit einem beliebigen Parameter. : {0}  
dany2={2}{0}{1}{1}{3}{4}
```

Quellcode:

```
import java.io.*;  
import java.text.MessageFormat;  
import java.util.*;  
  
public class PropertiesDemo {  
    public static void main(String[] args) {  
        Properties settings = new Properties();  
  
        try {  
            settings.load(new FileInputStream("Demo.properties"));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        // Umgang mit dynamischen Texten  
        MessageFormat nachricht = new MessageFormat(settings.getProperty("dyna"));  
        Object[] text = {"mein Text"};  
        System.out.println(nachricht.format(text));  
        nachricht = new MessageFormat(settings.getProperty("dany2")); // olha  
        Object[] text2 = {"o", "l", "h", "a", "!"};  
        System.out.println(nachricht.format(text2));  
    }  
}
```

Ergebnis (Ausgabe auf Konsole):

```
Text mit einem beliebigen Parameter. : mein Text  
Hallo!
```

# ResourceBundle (!)

# Internationalisierung über ResourceBundle

## Vorteile der ResourceBundle

- Ziel: Mehrsprachige Anwendungen sollen ermöglicht werden
- Texte sind abhängig von den benutzerspezifischen Einstellungen

## Konkrete Umsetzung

- Kapselung der Übersetzungen in speziellen Dateien (Namensgebung -> siehe nächste Folie)
- Zugriff auf die Übersetzungen aus der Java-Applikation über sogenannte Aliase (Schlüssel)
- optional: die jeweilige Sprach- und Ländereinstellungen können auch zur Laufzeit noch geändert werden

# Namensbildung für Bundle-Dateien

## Regeln

- alle Dateien enden auf .properties
- bundleName\_localeLanguage\_localeCountry\_localeVariant
- bundleName\_localeLanguage\_localeCountry
- bundleName\_localeLanguage
- bundleName\_defaultLanguage\_defaultCountry\_defaultVariant | bundleName\_defaultLanguage\_defaultCountry
- bundleName\_defaultLanguage
- bundleName

## Beispiel

- bundleName ist DemoBundle
- Sprach- und Ländereinstellung ist Deutschland
- DemoBundle\_de\_DE.properties



# Beispiel für den Umgang mit ResourceBundle

```
import java.util.*;

public class BundleDemo {
    public static void main(String[] args) {
        String bundleName = "DemoBundle";

        try {
            Locale.setDefault(Locale.CHINA);
            ResourceBundle bundle = ResourceBundle.getBundle(bundleName);
            System.out.println("China: " + bundle.getString("alias"));

            Locale.setDefault(new Locale("de"));
            bundle = ResourceBundle.getBundle(bundleName);
            System.out.println("Deutsch: " + bundle.getString("alias"));

            Locale.setDefault(Locale.ENGLISH);
            bundle = ResourceBundle.getBundle(bundleName);
            System.out.println("English: " + bundle.getString("alias"));

        } catch (MissingResourceException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Ergebnis (Ausgabe auf Konsole):

```
China: Dies ist der Standardtext.
Deutsch: Dies ist die deutsche Variante.
English: This is the english version.
```

# Beispiel ResourceBundle: DemoBundle

DemoBundle.properties (Default):

```
#DemoBundle.properties  
alias=Dies ist der Standardtext.
```

DemoBundle\_de.properties (Deutsch):

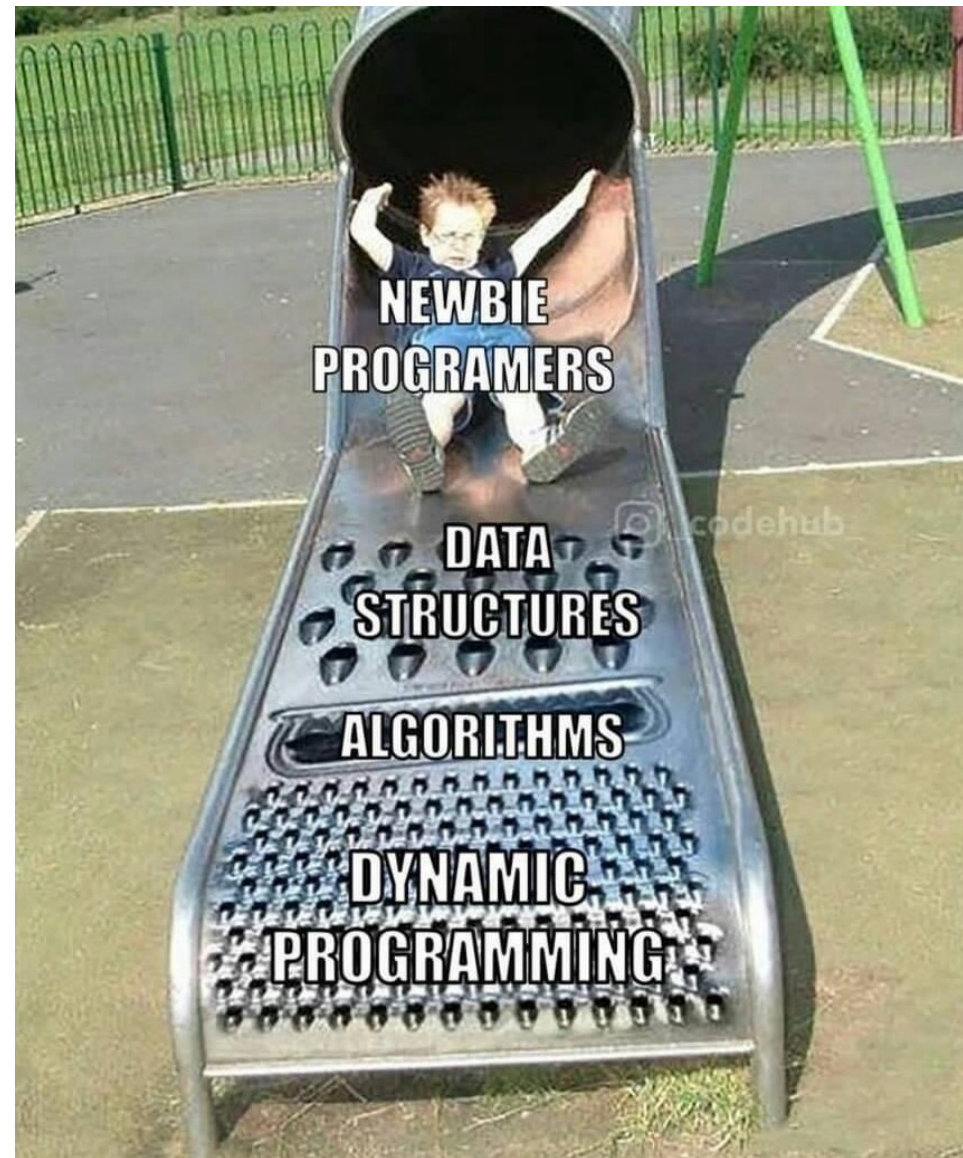
```
#DemoBundle_de.properties  
alias=Dies ist die deutsche Variante.
```

DemoBundle\_en.properties (English):

```
#DemoBundle_en.properties  
alias=This is the english version.
```

# Kapitel 12

# Datenstrukturen





Kylie Jenner 

@ikyliejenner

Can you guys please recommend books that made you cry?



Saransh Garg @saranshgarg

Replying to @ikyliejenner

**Data Structures and Algorithms in Java (2nd Edition)** 2nd E

# Kapitelübersicht - Programmieren 2

- 8. Exception Handling
- 9. Collection Framework
- 10. Swing
- 11. Optional: Input- & Output-Streams
- 12. **Datenstrukturen**
- 13. Algorithmen

# Lernziele

- Sie können die Eigenschaften, den internen Aufbau und wichtigsten Funktionen von folgenden Datenstrukturen nennen und implementieren
  - Verkettete Liste / Doppelt verkettete Liste (Linked List)
  - Stapel (Stack)
  - (Warte-)Schlange (Queue)
  - Binärbaum (Binary Tree)
- Sie können Iteration und Rekursion unterscheiden und passend anwenden
- Sie können Angaben zu Komplexitäten von Algorithmen (Funktionen) auf den Datenstrukturen machen

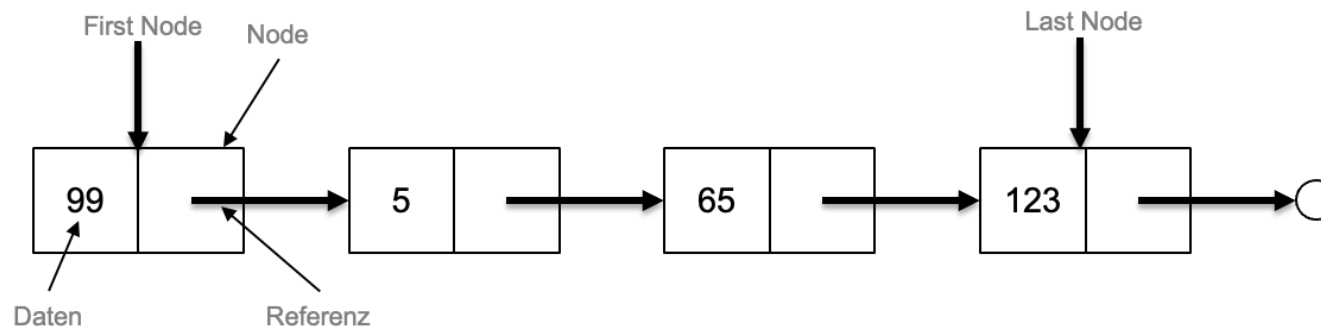
Listen



# Verkettete Liste (Linked List)

## Methoden

- add() / addFirst() / addLast() / addAfter()
- remove(by object / by search attributes)
- find(by search attributes)
- contains(by object)
- getFirst() / getLast()
- ...

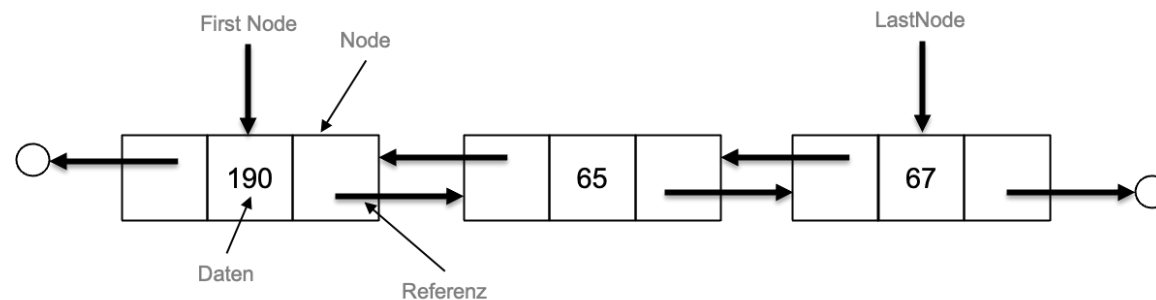


# Doppelt Verkettete Liste (Double Linked List)

## Methoden

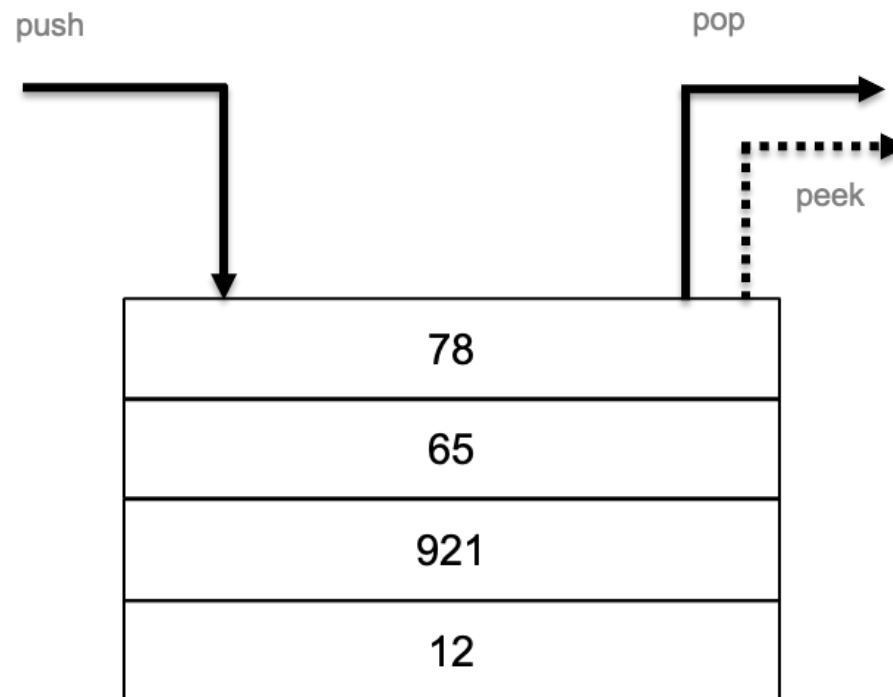
- add() / addFirst() / addLast() / addAfter() / addBefore()
- remove(by object / by search attributes)
- find(by search attributes)
- contains(by object)
- getFirst() / getLast()
- ...

Parallel Suche:  $O(n/2) \rightarrow O(n)$



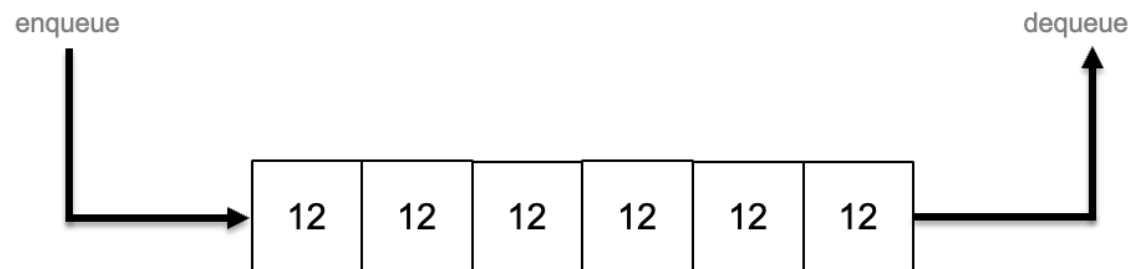
# Stapel (Stack)

- Eigenschaften
  - Ein-/Ausgabe Strategie: LIFO (Last In - First Out)
    - Neue Elemente werden oben auf den Stapel gelegt, das nächste Element (neueste auf dem Stapel) wird oben vom Stapel entnommen
- Methoden
  - push()
  - pop()
  - peek()

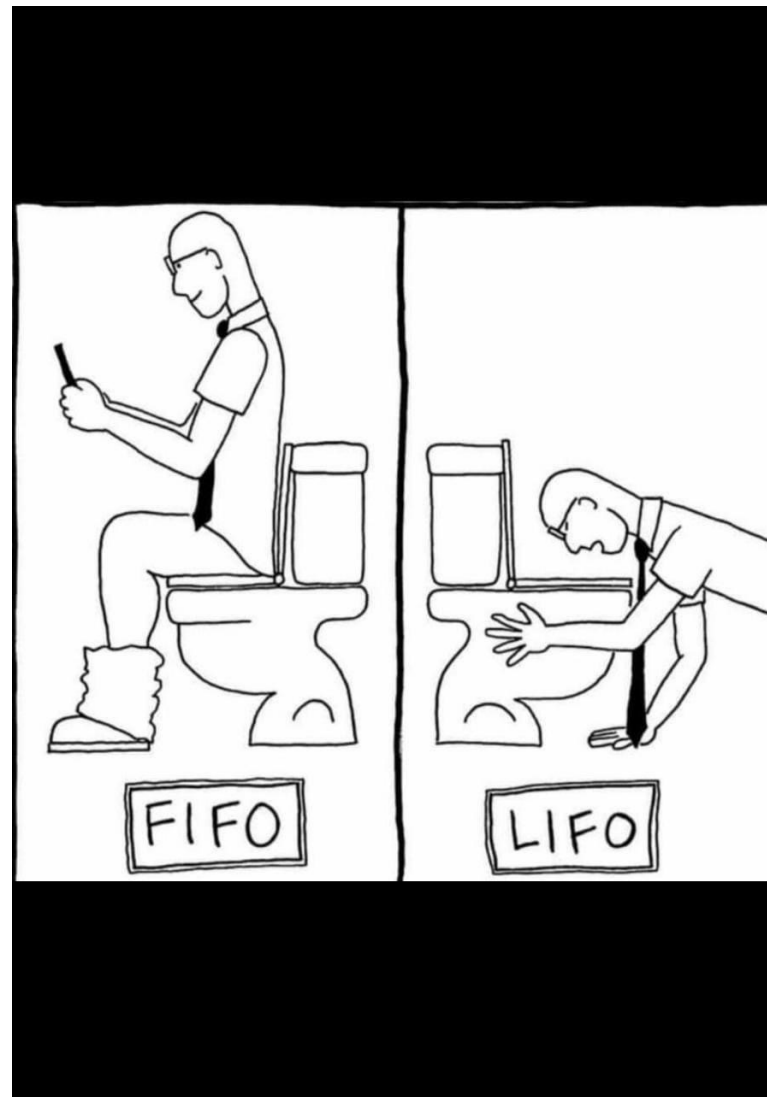


# (Warte-) Schlange (Queue)

- Eigenschaften
  - Ein-/Ausgabe Strategie: FIFO (First In - First Out)
    - Neue Elemente werden in die Schlange eingereiht, das älteste Element (am längsten in der Schlange) wird als nächstes entnommen
- Methoden
  - enqueue()
  - dequeue()



# Ein-Ausgabe Strategien - endlich erklärt

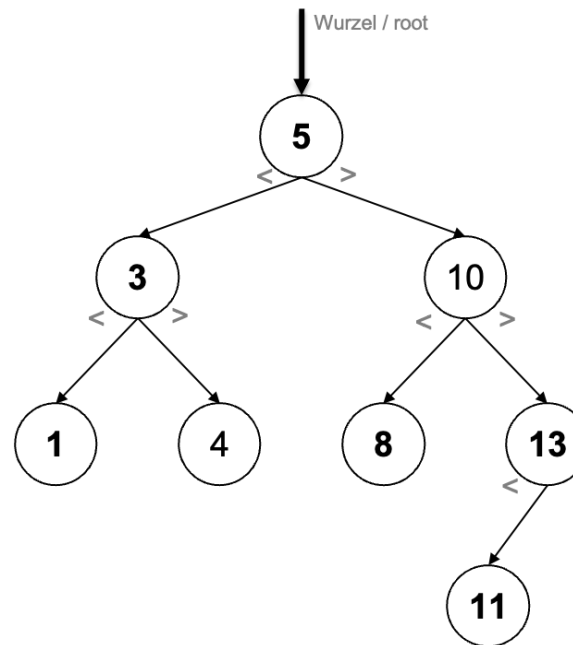


# Bäume

# Binär-Baum / Binäre-Such-Bäume (Binary Tree)

## Methoden

- insert(object)
- [find(by search attribute)] (Nicht für generische Container)
- contains(object)
- Remove(object)
- ...



TODO: ALVTree

- ALV -> Adelson-Velsky and Landis (bennant nach den Erfindern)
- selbstständig balancierende Binäre-Such-Bäume
- Visualisation



# Iteration VS. Rekursion

- Sowohl Iteration als auch Rekursion werden für sich wiederholende Anweisungen in Algorithmen verwendet
- Iteration nutzt Schleifen zur Aneinanderreihung
- Rekursion nutzt Methoden/Funktionen zur Schachtelung
- Beide Arten benötigen definierte Abbruchkriterien

## Eigenschaften von rekursiver Programmierung

- belegt mit jeder neu geschachtelten Ebene extra Speicher auf dem Stack (Arbeitsspeicherbereich)
- jeder (Methoden-/Funktions-)Aufruf für eine neue Ebene braucht extra Rechenleistung (Zeit / Performance)
- -> intensiver im Speicher- und Rechenleistungsverbrauch
- Quellcode bleibt (im speziellen in komplexen Fällen) übersichtlicher

# Komplexitäten bei Algorithmen

- Komplexitätsangaben können für verschiedene Dimensionen angegeben werden (z.B. Zeit, Speicher, ...)
- wir betrachten Laufzeitkomplexitäten (Dimension: Zeit) in Abhängigkeit von Mengen
- Angabe in der O-Notation ->  $O(f(n))$
- Betrachtung von Algorithmen abhängig von Datenkonstellationen
  - Best-Case: Laufzeit im besten Fall
  - Average-Case: Laufzeit im Mittel
  - Worst-Case: Laufzeit im schlechtesten Fall
  - Annahme: Best- und Worst-Case sind Randfälle
- Gängigste Laufzeitkomplexitäten:

O-Notation	Bezeichnung	: -)
$O(1)$	konstant	$O(\text{yeah})$
$O(\log n)$	logarithmisch	$O(\text{nice})$
$O(n)$	linear	$O(k)$
$O(n^2)$	quadratisch	$O(\text{my})$
$O(d^n)$ für $d > 1$	exponential	$O(\text{no})$
$O(n!)$	faktoriell	$O(\text{mg!})$

# Kapitel 13

# Algorithmen

# Kapitelübersicht - Programmieren 2

- 8. Exception Handling
- 9. Collection Framework
- 10. Swing
- 11. Optional: Input- & Output-Streams
- 12. Datenstrukturen
- 13. **Algorithmen**