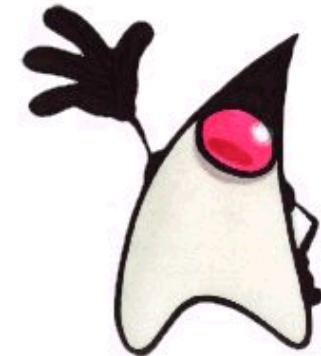
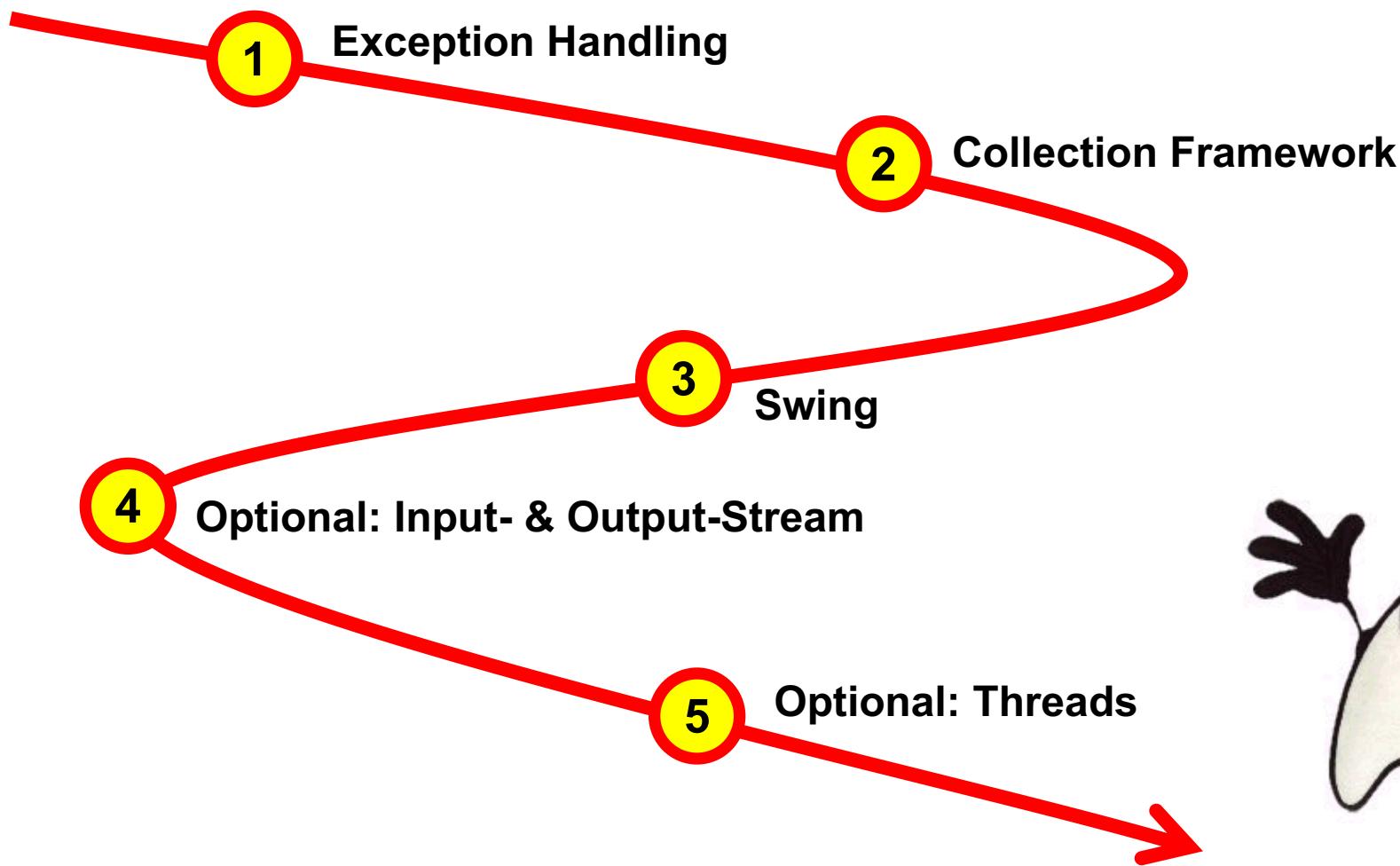


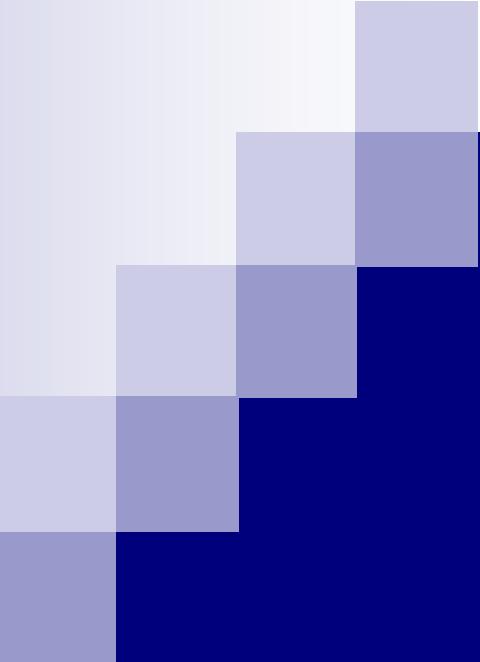
# Fortgeschrittene Programmierung





# Themenüberblick

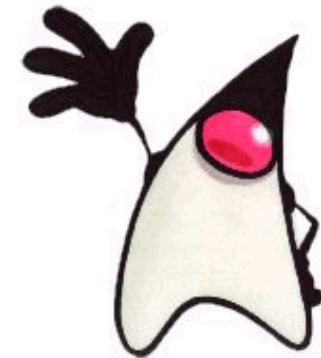
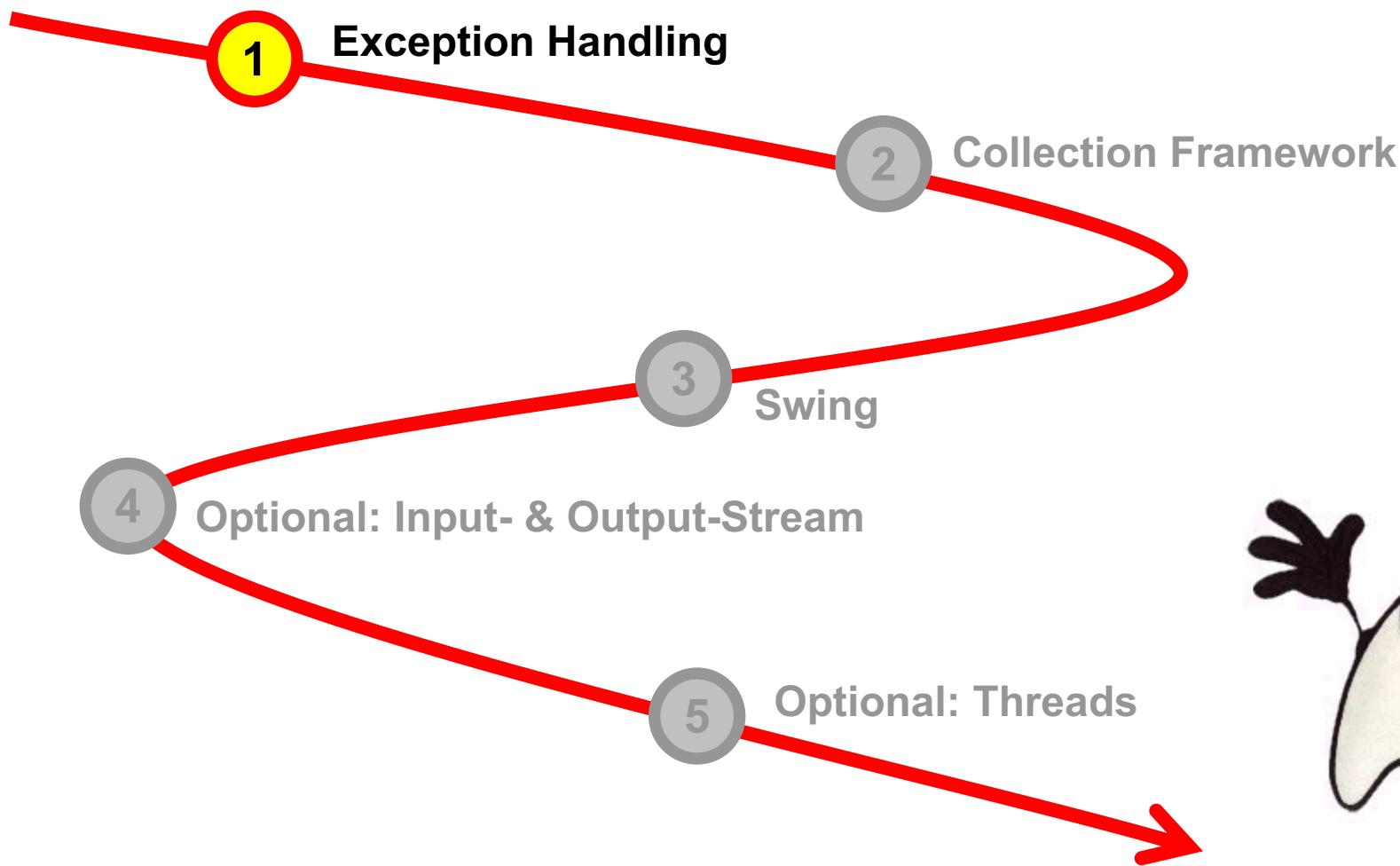




# Programmierung 2

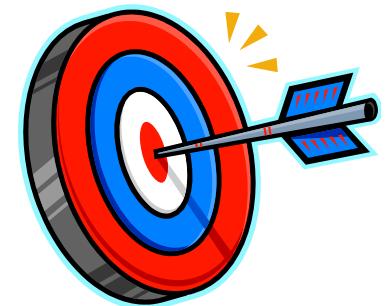
## Kapitel 1 Exception Handling

# Themenüberblick



# 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



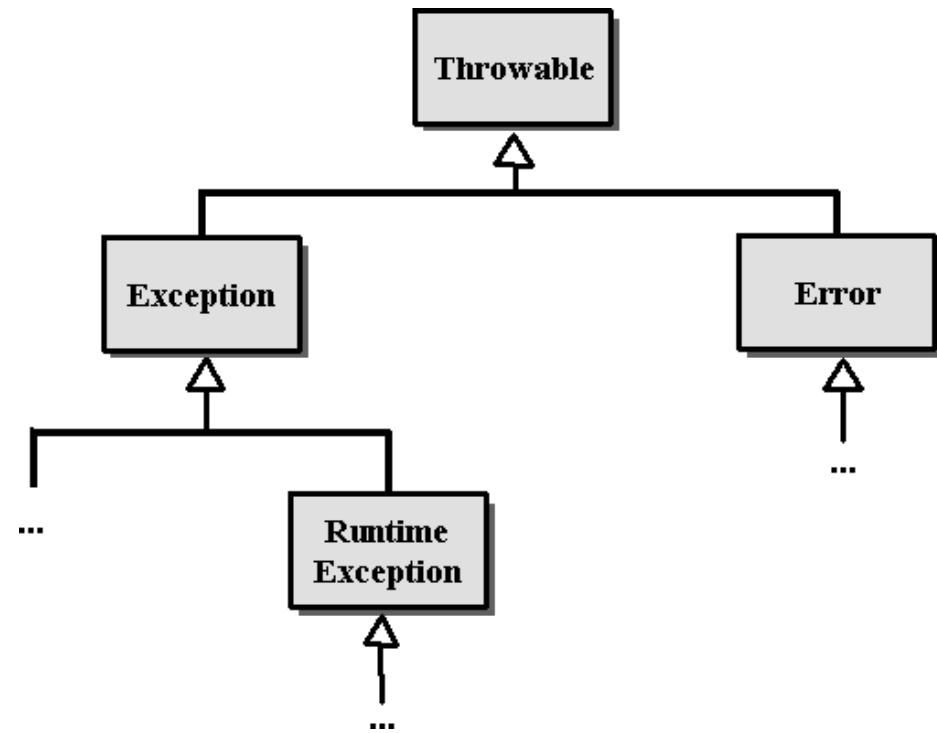
# Fehler in Java

## Compiler-Fehler

- syntaktische Fehler werden beim Kompilieren erkannt

## Laufzeitfehler

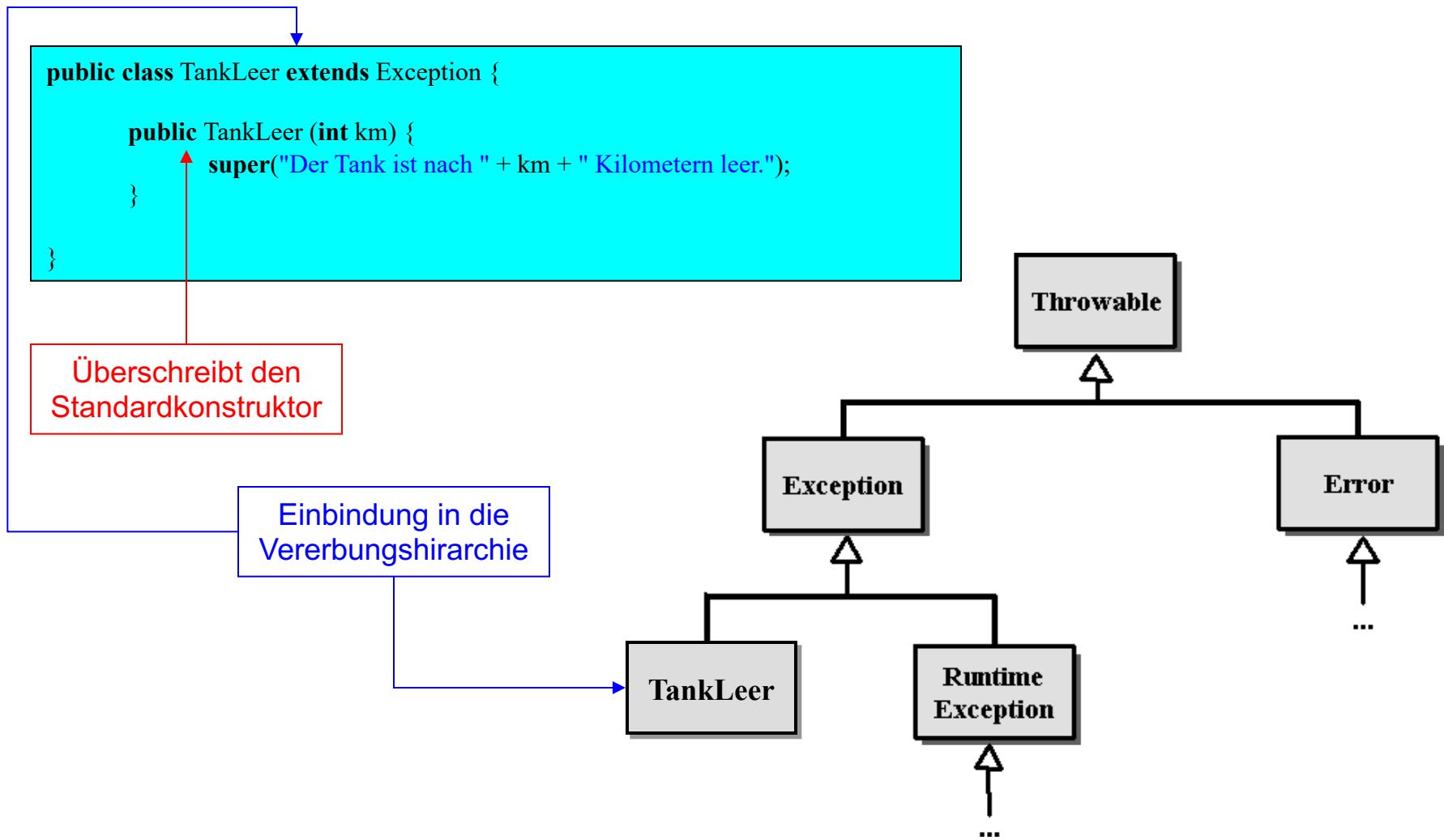
- Fehler (Error) sollte nicht behandelt werden
- Ausnahmen (Exceptions)
  - ◆ Exception muss behandelt werden
  - ◆ RuntimeException kann behandelt werden



# 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)

# Erzeugen eigener Ausnahmeklassen



# 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);  
            }  
        }  
    }  
}
```

Auslösen einer Ausnahme mit *throw*

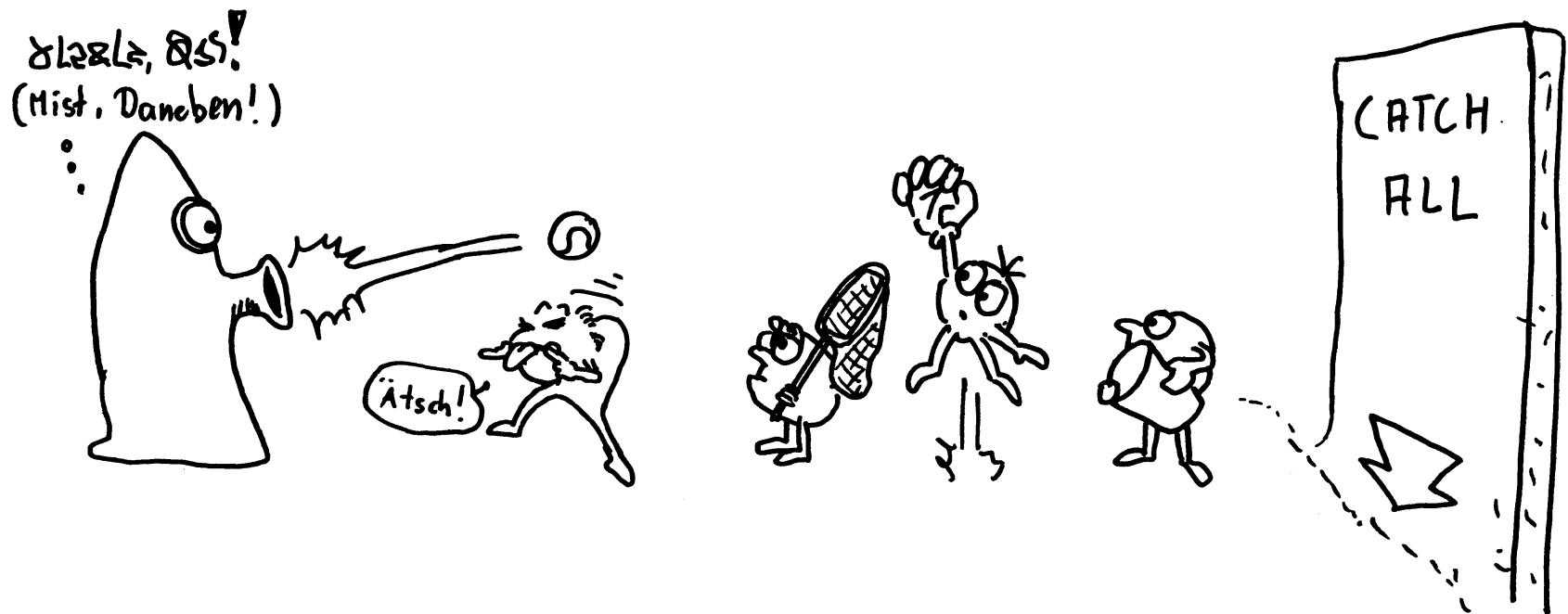
Erzeugt ein Objekt der Klasse *TankLeer*

*throws* definiert alle Ausnahmen, die Auftreten können

Im Ausnahmefall erfolgt die Weitergabe an den Aufrufer

Alternativ zu *throws* könnte die Ausnahme auch behandelt werden

# Ausnahmen behandeln



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

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

# Ausnahmen behandeln

try markiert den Beginn  
des Über-  
wachungsbereichs

```
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());  
        }  
        ...  
    }  
}
```

catch beendet die  
Überwachung und  
fängt mögliche  
Ausnahmen 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

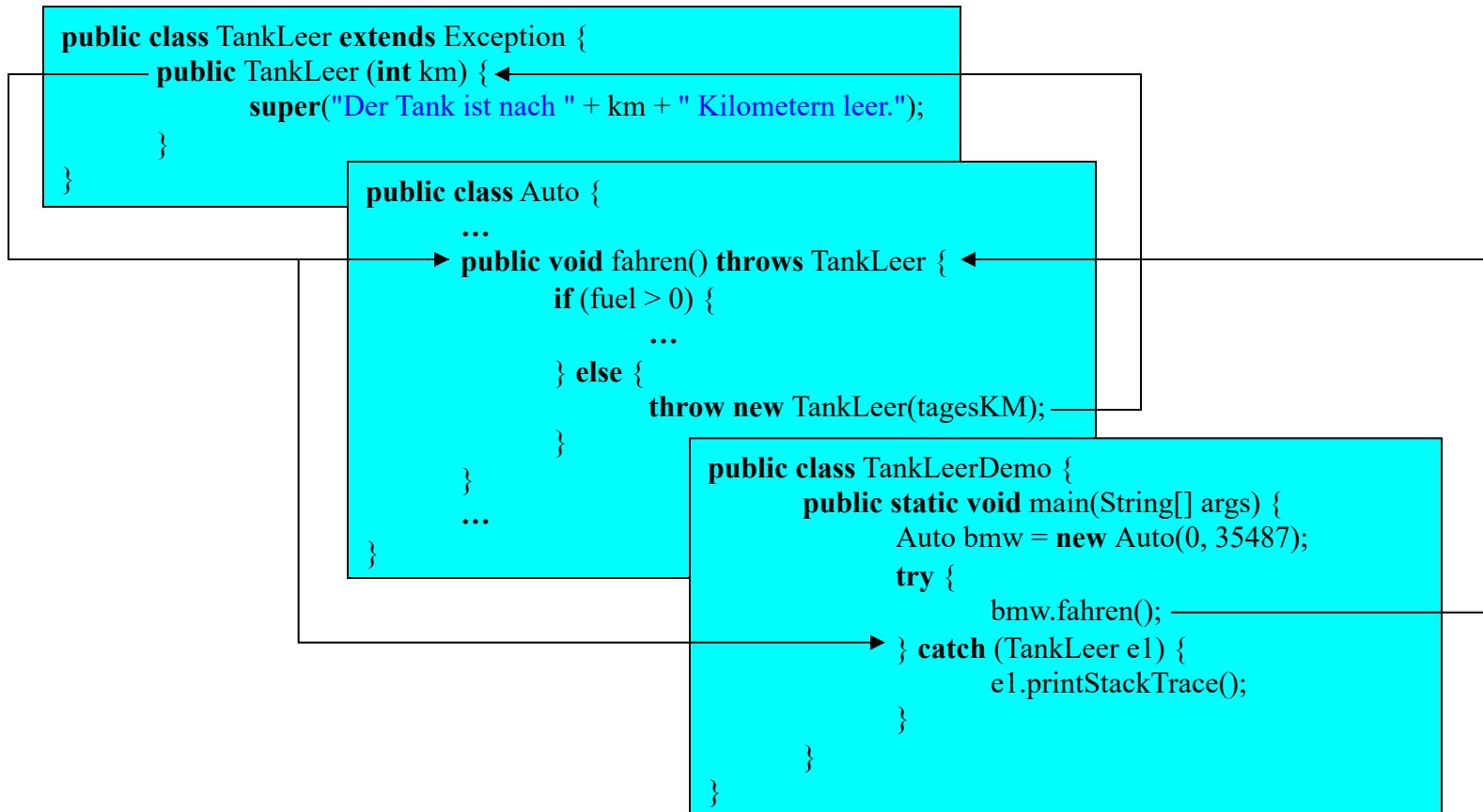
```
prog2.demos.exceptions.TankLeer: Der Tank ist nach 1100  
Kilometern leer.
```

- 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 Exceptions

- müssen mit try/catch abgefangen oder mit throws an den Aufrufer weitergegeben werden
- checked Exceptions treten nur durch explizites Auslösen auf



# Unchecked Exceptions

- treten erst zur Laufzeit auf
- werden automatisch an den Aufrufer weitergegeben
- können ebenfalls mit try/catch aufgefangen werden
- oftmals handelt es sich um logische Programmierfehler

```
public class DivisionNull {  
  
    public static void main(String[] args) {  
  
        int zahl1 = 10;  
        int divisor = 0;  
        double ergebnis = 0;  
  
        try {  
            ergebnis = zahl1 / divisor;  
        } catch (ArithmeticalException ae){  
            System.out.println(ae.getMessage());  
        }  
        System.out.println(ergebnis);  
    }  
}
```



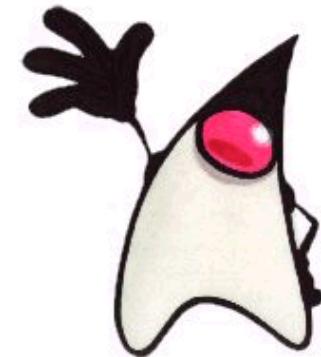
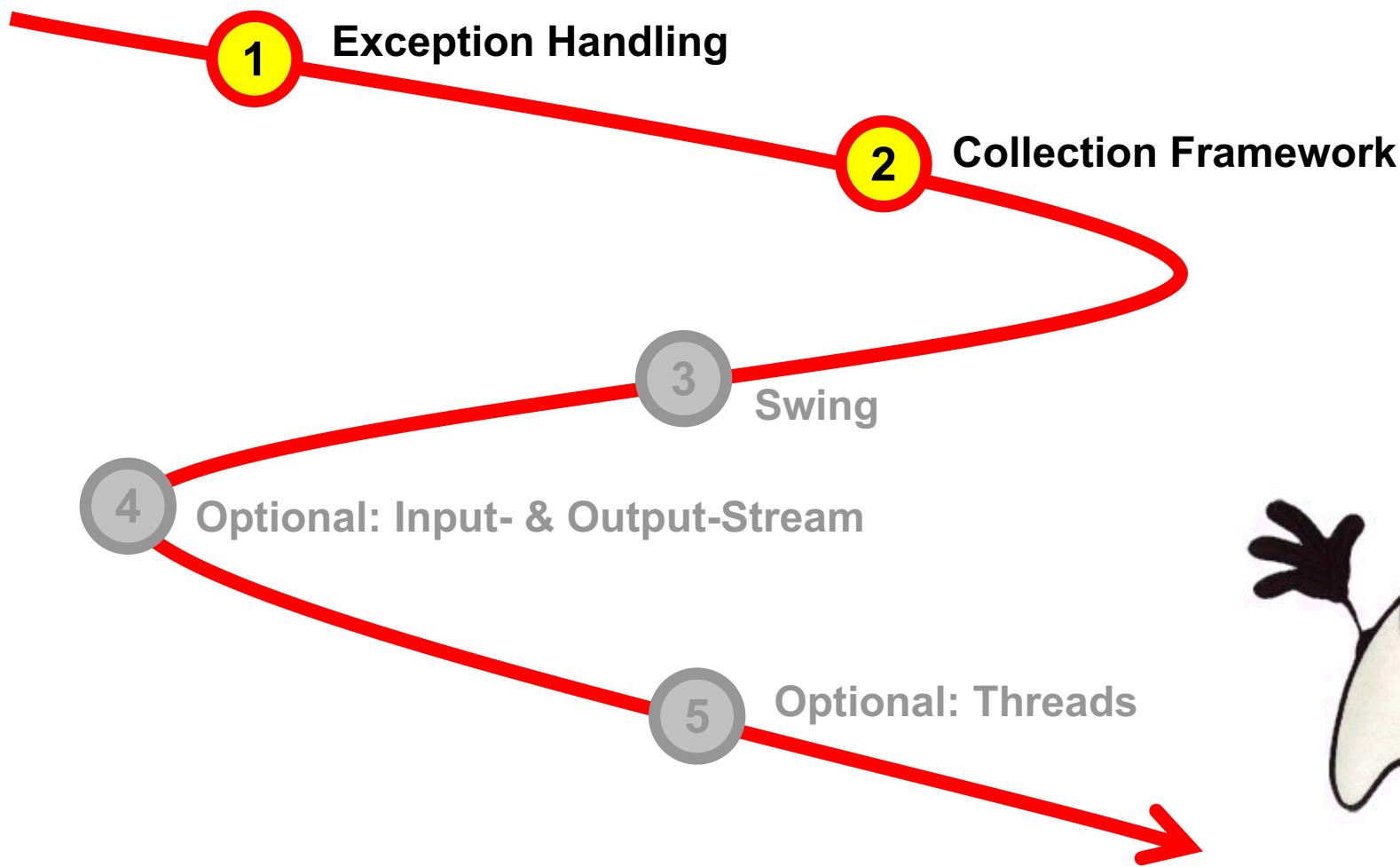
RuntimeException



# Programmierung 2

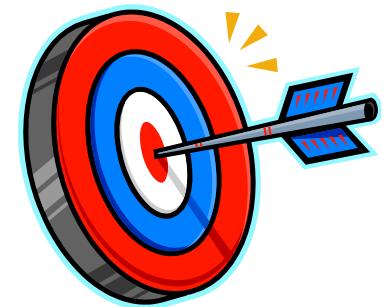
Kapitel 2  
Collection Framework

# Themenüberblick

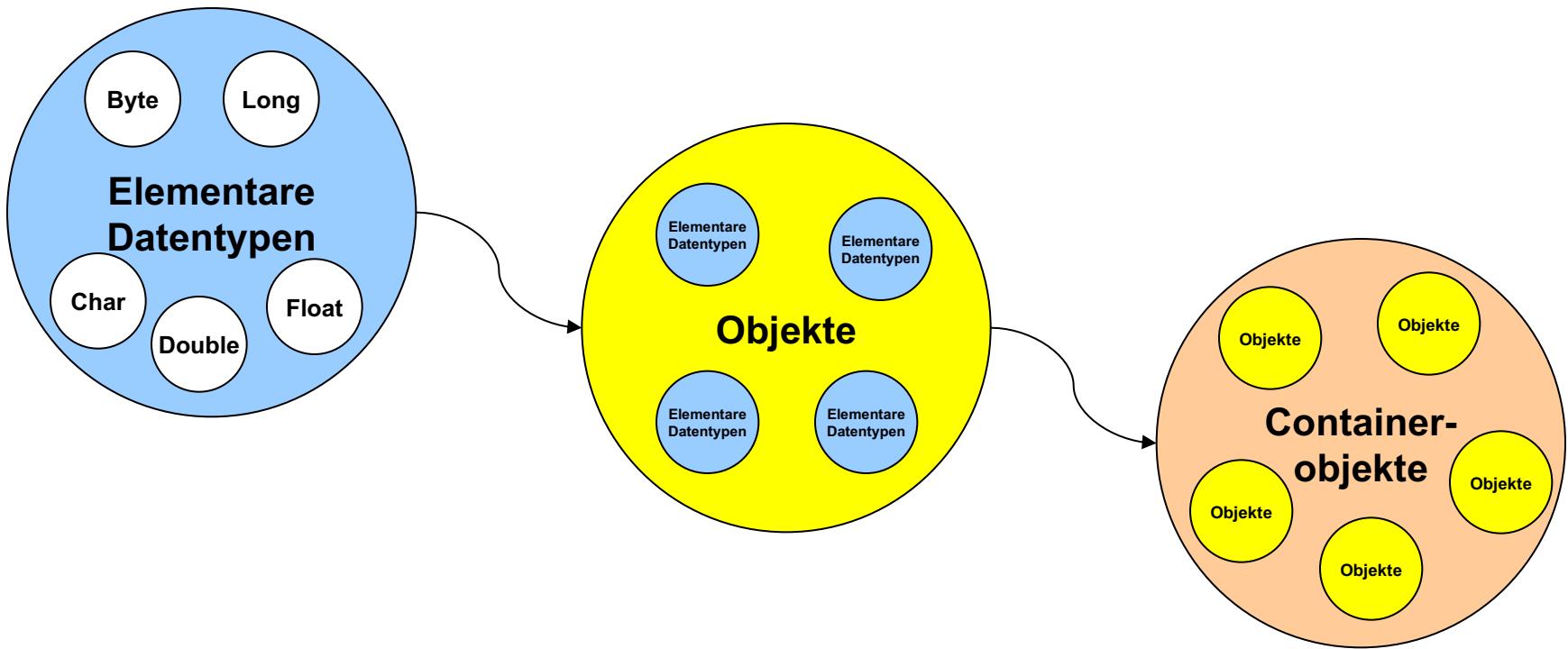


# Lernziele

- 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



# Datenstrukturen und -container



Das Collection Framework bietet generische Container

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

# Die drei Arten von Containern

## Listen (List)

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

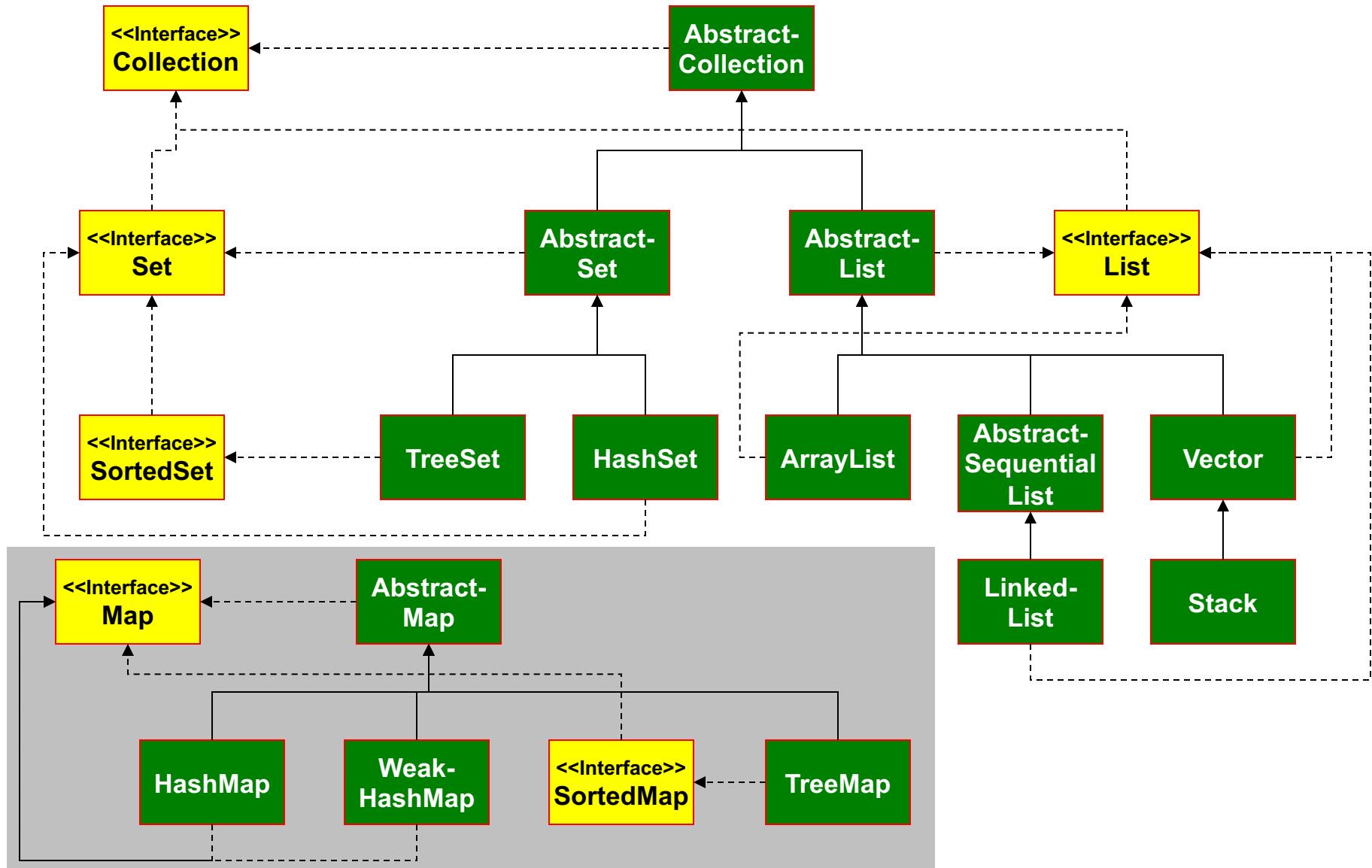
## Mengen (Set)

- 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

# Überblick über das Collection Framework



# 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
- Vector und ArrayList sind 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
  - ◆ bis Java 5 nur „vorwärts“ durchlaufen werden
  - ◆ ab Java 6 in beide Richtungen 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 Liste mit Iteratoren

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

public class DemoListe {
    public static void main(String[] args) {
        ArrayList liste = new ArrayList();
        liste.add("Otto");
        liste.add("Karl");
        liste.add("Ludwig");
        liste.add(new Auto(0, 0));
        liste.add(2,"Otto");
        liste.set(3,"Überschreibt den Ludwig");
        System.out.println(liste.contains("Otto"));
        System.out.println(liste.indexOf("Ludwig"));
        System.out.println(liste.get(3));
        System.out.println(liste.size());
    }
}
```

erzeugt den Iterator  
für das Listenobjekt  
über die Methode  
*iterator()*

überprüft mit  
*hasNext()*, ob es im  
Listenobjekt noch ein  
weiteres Element gibt

*next()* besorgt sich  
das nächste Objekt  
aus der Liste

true  
-1  
Überschreibt den Ludwig  
5  
Otto  
Karl  
Otto  
Überschreibt den Ludwig  
prog2.demos.exceptions.Auto@923e30  
0

# 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)
- 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

# 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 Rückgabewerte
  - ◆ 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

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

public class Menge {
    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());
        }
    }
}
```

```
public class Student implements Comparable {
    private String vorname;
    private String nachname;
    private int matrikelNo;
    public Student(String vName, String nName, int no) {
        this.vorname = vName;
        this.nachname = nName;
        this.matrikelNo = no;
    }
    ...
    public int compareTo(Object vStudent) {
        return this.matrikelNo - ((Student) vStudent).getMatrikelNo();
    }
}
```

19853 Karl Schmidt
65871 Hans Müller
65872 Hans Müller
75382 Peter Maier

# 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
- 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)`
- 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;
    }
}
```

```
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());
        }
    }
}
```

```
85383 Karl Maier
75382 Peter Maier
65871 Hans Müller
65872 Hans Müller
19853 Karl Schmidt
```

# 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

# 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`-Vertrag 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

```
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;  
    }  
}
```

```
public class DemoEquals1 {  
    public static void main(String[] args) {  
        Haustier tier1 = new Haustier("Hase", 10);  
        Haustier tier2 = new Haustier("Hase", 10);  
        System.out.println("Tier1 verglichen mit Tier2 über den ===-Operator");  
        System.out.println(tier1 == tier2);  
        System.out.println("... und der Vergleich mit der equals()-Methode");  
        System.out.println(tier1.equals(tier2));  
    }  
}
```

Tier1 verglichen mit Tier2 über den ===-Operator  
false  
... und der Vergleich mit der equals()-Methode  
true

# Das Überschreiben der equals()-Methode

```
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;  
    }  
}
```

```
public class DemoEquals2 {  
    public static void main(String[] args) {  
        Hund hund1 = new Hund(20, "Collie");  
        Hund hund2 = new Hund(50, "Bernhardiner");  
        System.out.println("Hund1 verglichen mit Hund2 über den ===-Operator");  
        System.out.println(hund1 == hund2);  
        System.out.println("... und der Vergleich mit der equals()-Methode");  
        System.out.println(hund1.equals(hund2));  
    }  
}
```

Hund1 verglichen mit Hund2 über den ===-Operator  
false  
... und der Vergleich mit der equals()-Methode  
false

# Zusammenhang hashCode() und equals()

- Verwendung für die Verwaltung der Einträge in hash-basierten Datencontainern (HashSet, HashMap, ...)
- korrekte Verwaltung der Einträge basiert auf folgender Bedingung
  - ◆ 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

# Das Überschreiben der hashCode()-Methode

```
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;  
    ...  
    // Getter- und Setter-Methoden  
    public boolean equals(Object objekt) {  
        ...  
    }  
  
    public int hashCode() {  
        return super.hashCode() ^ this.getRasse().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())

```
public class Haustier {  
  
    public int hashCode() {  
        int hc = 17;          // beliebiger Initialwert  
        int hashMultiplier = 59; // beliebige (kleine) Primzahl  
  
        hc = hc * hashMultiplier + <feldwert>  
        ...  
  
        return hc;  
    }  
}
```

# 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

# 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

# Beispiel für eine TreeMap mit Iteratoren

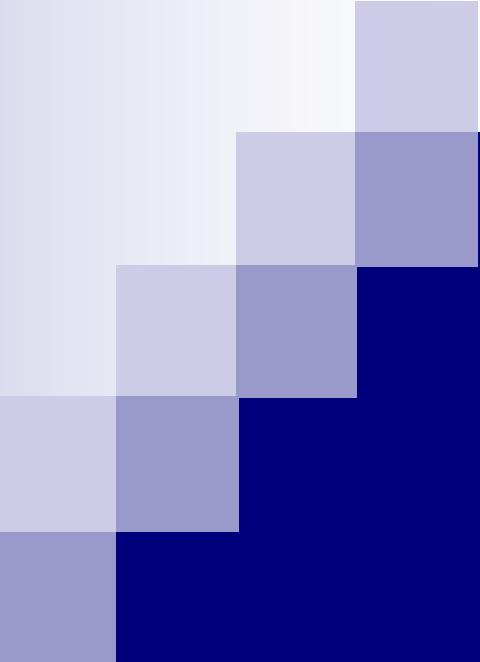
```
import java.util.Set;
import java.util.Iterator;
import java.util.TreeMap;

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());
    }
}
```

beschafft sich das  
Schlüssel-Set und  
erzeugt einen  
passenden Iterator

liest den nächsten  
Schlüssel aus und  
beschafft sich den  
dazugehörigen Wert

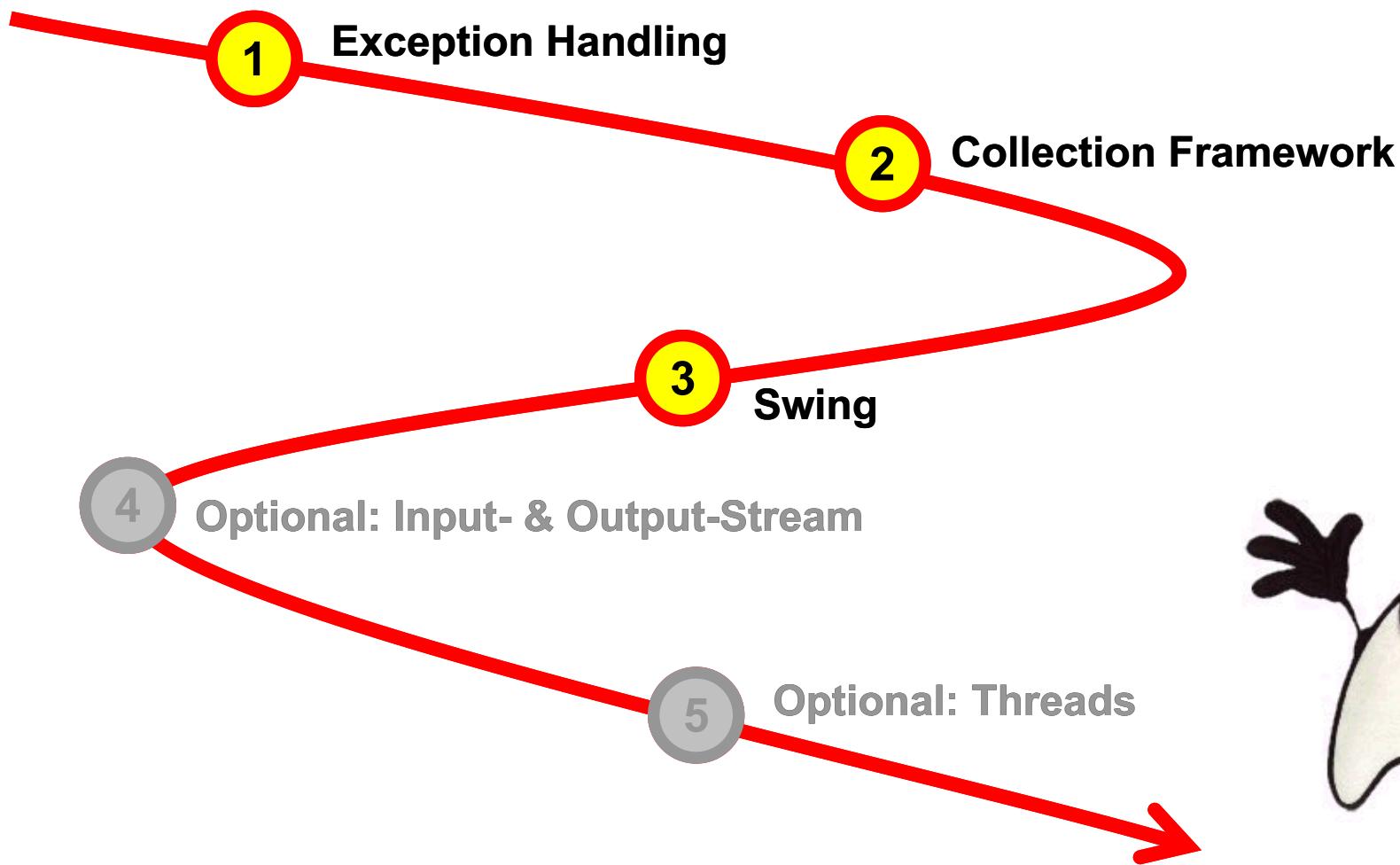
```
Schlüssel: 100 Wert: Labrador
Schlüssel: 110 Wert: Bernhardiner
Schlüssel: 120 Wert: Schäferhund
Schlüssel: 130 Wert: Cocker
4
```



# Programmierung 2

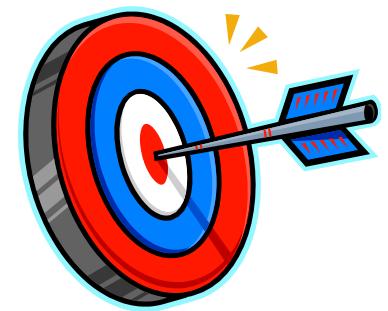
## Kapitel 3 Swing

# Themenüberblick



# Lernziele

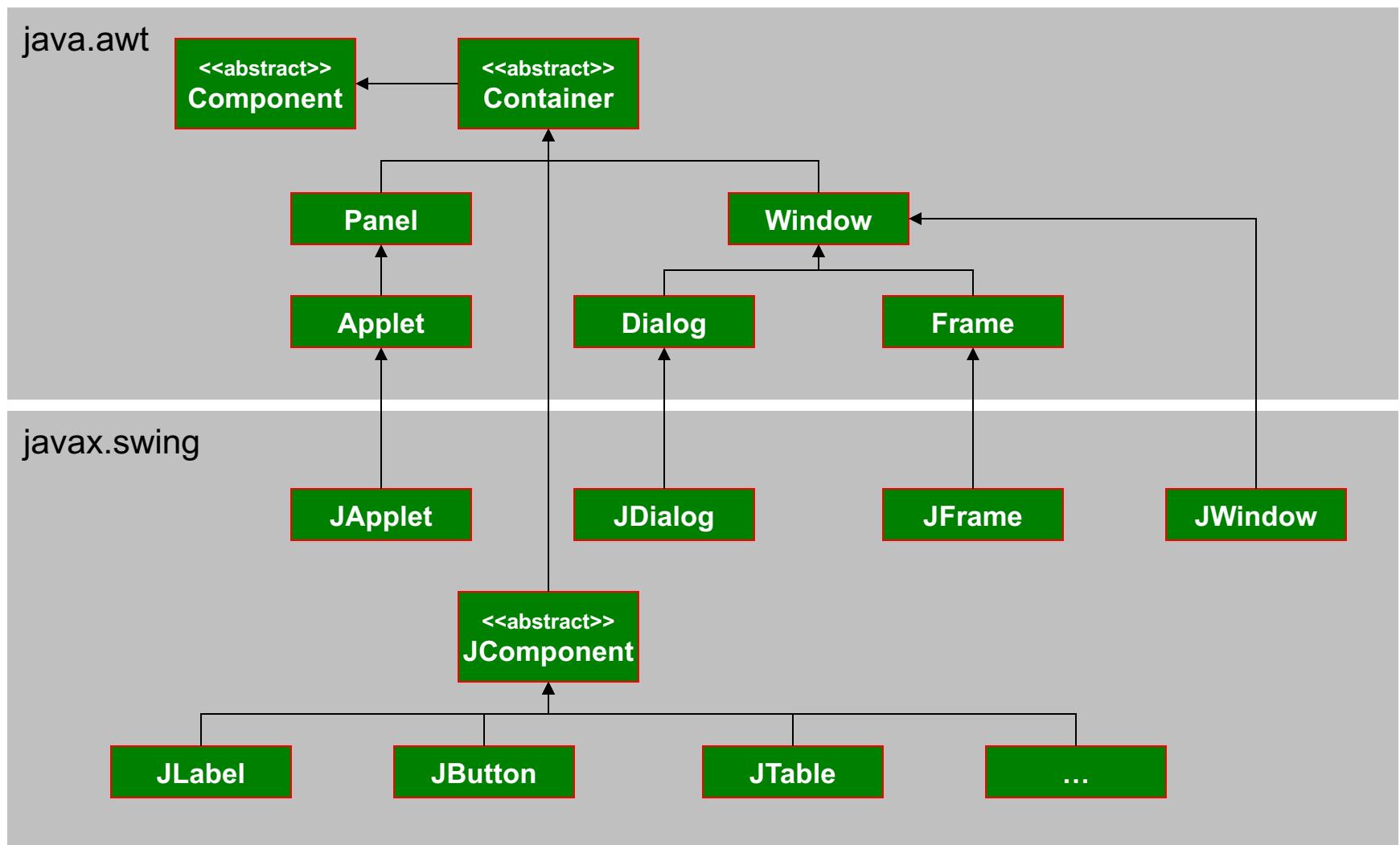
- Sie können den wesentlichen Unterschied zwischen AWT und Swing erläutern
- Sie können mit Swing einfache Fenster erzeugen und schließen
- Sie können unterschiedliche Layouts in Verbindung mit Panels einsetzen
- Sie können einfache Benutzerdialoge mit ausgewählten Swing-Komponenten erstellen
- Sie können 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



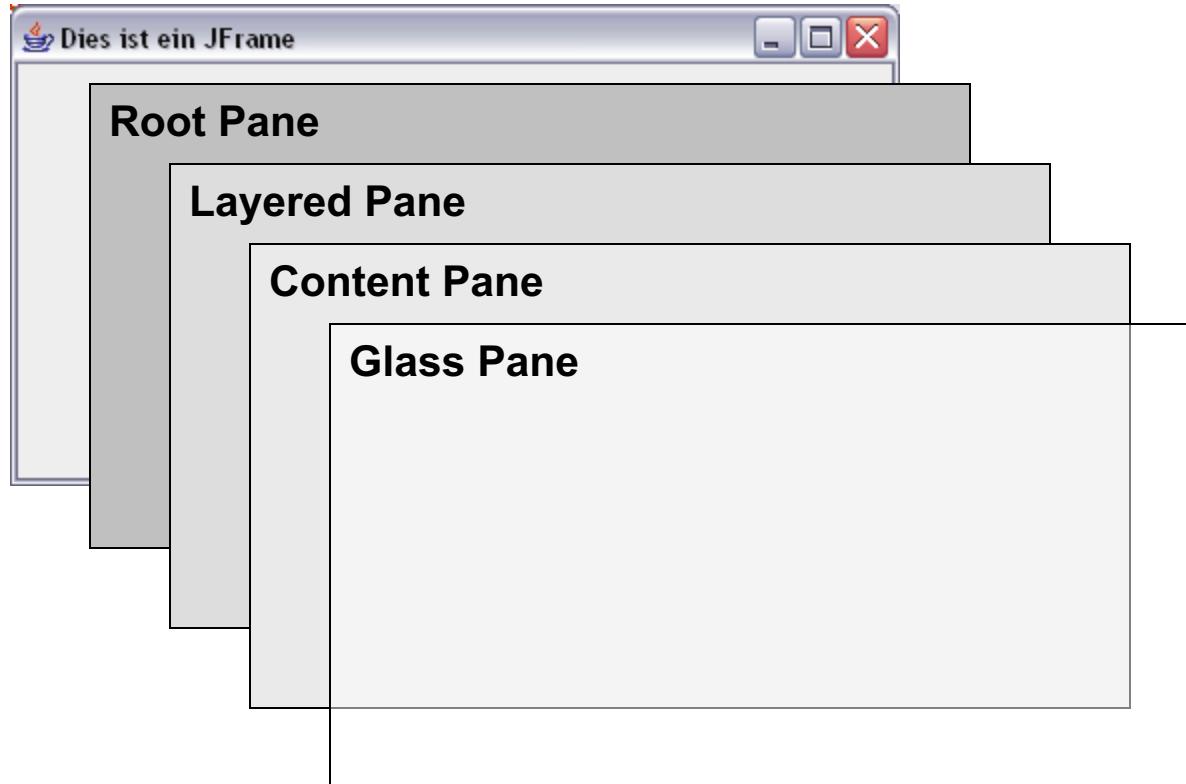
# Abgrenzung von AWT und Swing

- 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



# 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

# 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

# Layouts im Rahmen von Swing

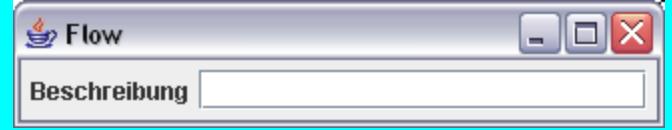
- Anordnung der Elemente eines Containers nach bestimmten Verfahren über Layout-Manager
- wesentliche Layout-Manager
  - ◆ FlowLayout ordnet seine Elemente von links nach rechts
  - ◆ BorderLayout ermöglicht eine Anordnung in 5 verschiedenen Bereichen (NORTH, EAST, SOUTH, WEST und CENTER)
  - ◆ GridLayout 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



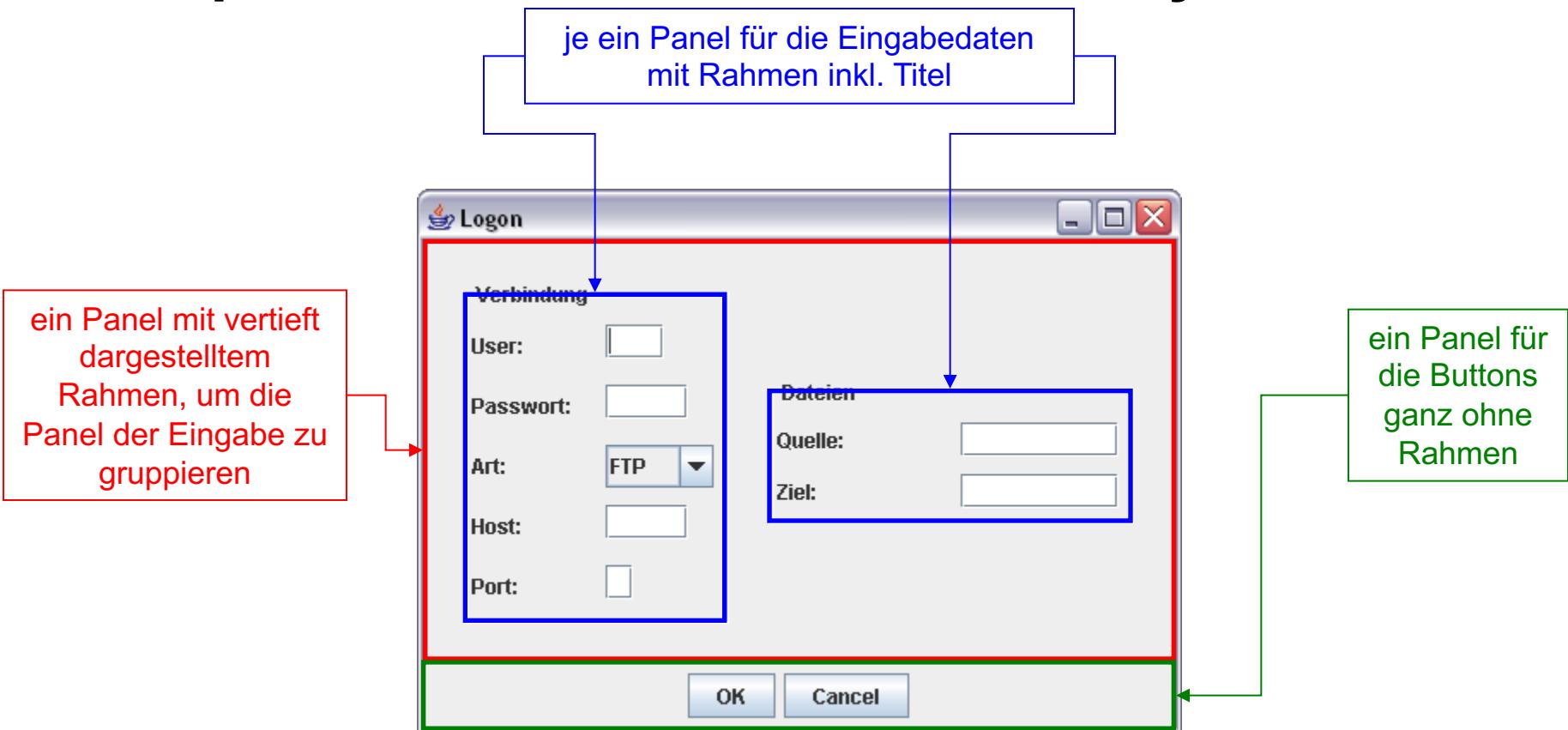
# Der Container JPanel

- 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

```
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

# Panels mit Rahmen hervorheben

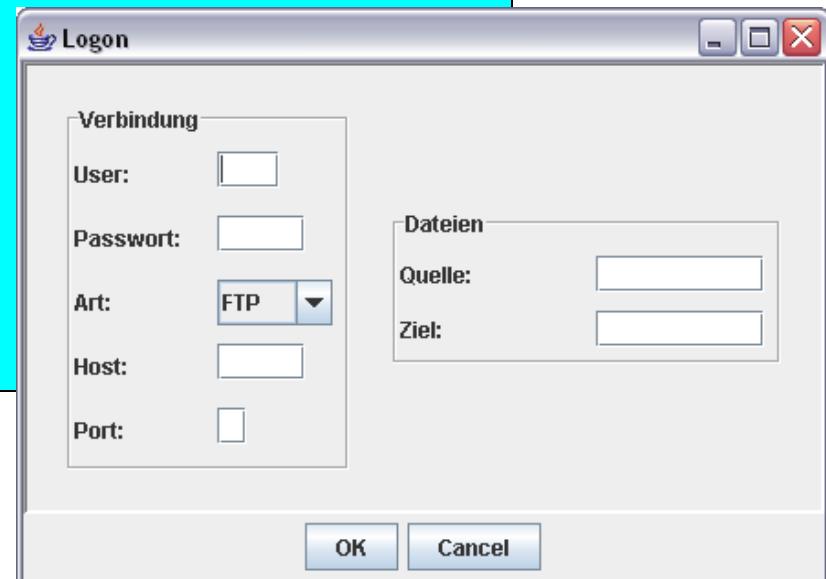
- Rahmen sind über Klassen realisiert, die das Interface Border implementieren
- Rahmen sollten nicht direkt über die Konstruktoren der Rahmen-Klassen sondern über die Klassenmethoden der BorderFactory erzeugt werden
- jeder Swing-Komponente kann mit der Methode setBorder(Border b) ein Rahmen zugewiesen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

(Quelle: Java ist auch eine Insel, 5.Auflage)

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



# 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.setLayout(new GridLayout(2,1));
        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 boolesche 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

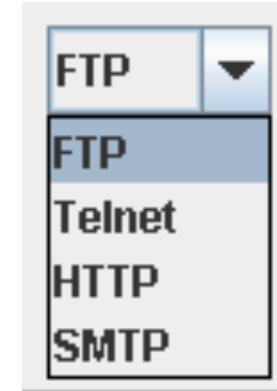
# Validierende Textfelder als spezielle Form

- 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



# 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 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.getSelectedIndex().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();
    }
}
```

importiert die wesentlichen Klassen ItemEvent und ItemListener

implementiert das Interface ItemListener mit seiner Methode itemStateChanged() in einer anonymen Klasse

erzeugt eine JComboBox und ordnet ihr den zuvor implementierten ItemListener zu

# 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 Erkennungsstring, 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() {
        ...
        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();
    }
}
```

importiert die wesentlichen Klassen ActionEvent, ActionListener und JButton

implementiert das Interface ActionListener mit seiner Methode actionPerformed() in einer anonymen Klasse

erzeugt zwei Buttons und ordnet sie dem zuvor implementierten ActionListener zu

# 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();  
    }  
}
```

importiert die wesentlichen Klassen ItemEvent und ItemListener

implementiert das Interface ItemListener mit seiner Methode itemStateChanged()

erzeugt eine JCheckBox und ordnet ihr den zuvor implementierten ItemListener zu



# 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();
    }
}
```

importiert die wesentlichen Klassen ActionEvent und ActionListener

implementiert das Interface ActionListener mit seiner Methode actionPerformed()

erzeugt zwei RadioButtons und ordnet sie einer ButtonGroup sowie dem ActionListener zu

# 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);
    }
    ...
}
```

importiert die wesentlichen Klassen ActionEvent, ActionListener und Menüklassen

implementiert das Interface ActionListener mit seiner Methode actionPerformed()

erzeugt die Menüleiste mit Menüs und Menüeinträgen, ordnet die Menüeinträge dem ActionListener und die Menüleiste dem JFrame zu

# 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(zuhorer);
        ok.setToolTipText("Führt die Funktion aus");
        ...
    }
    public static void main(String[] args) {
        DemoToolTip fenster = new DemoToolTip();
    }
}
```



# Weitere Klassen aus dem Package Swing

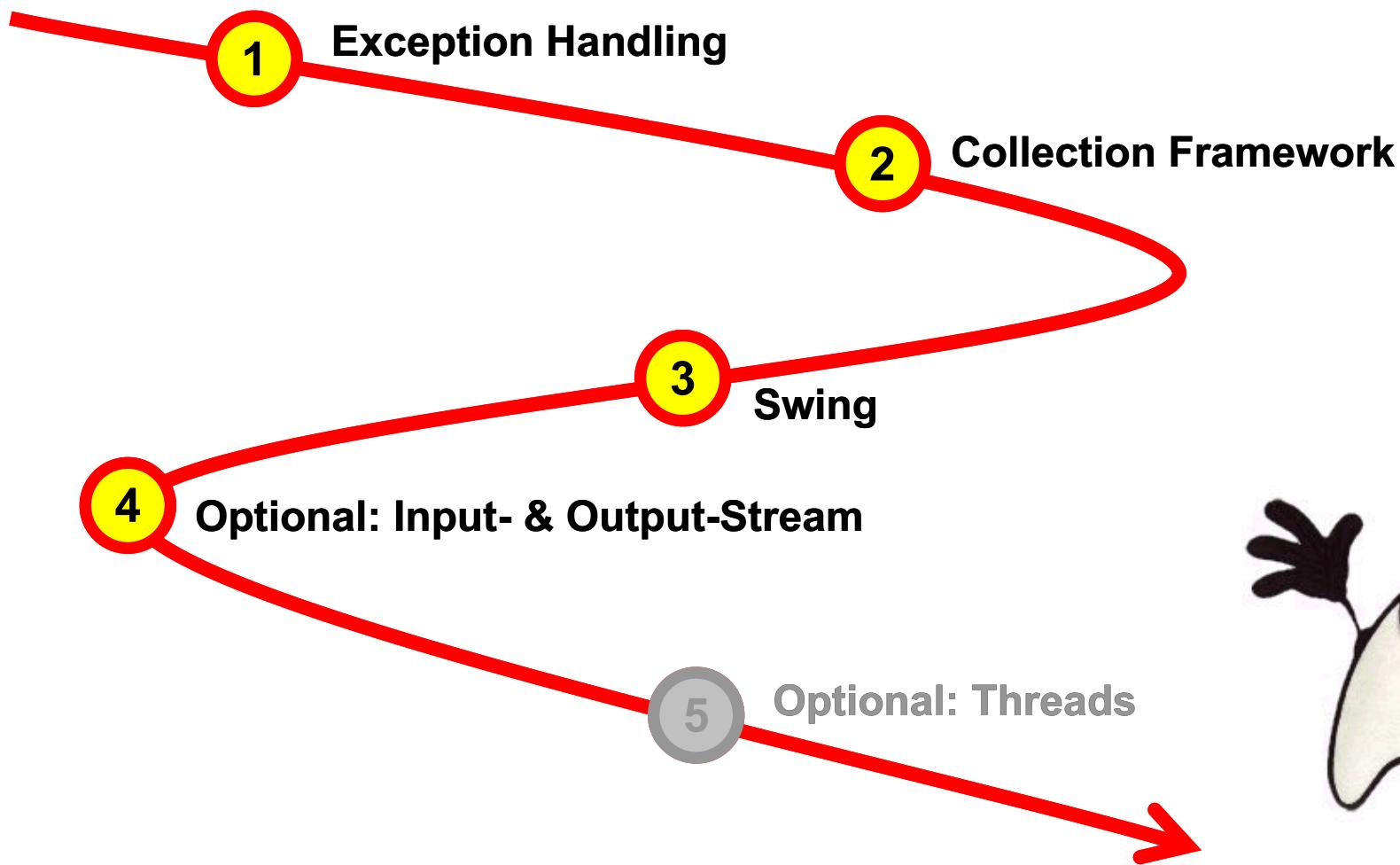
- **JTable**  
dient der Erstellung zweidimensionaler Tabellen
- **JTree**  
ermöglicht die Darstellung von Bäumen ähnlich dem Windows Explorer bestehend aus Knoten und Blättern
- **JToolBar**  
dient der Erstellung von Symbolleisten analog den Microsoft Office-Produkten
- **JColorChooser**  
dient der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



# Programmierung 2

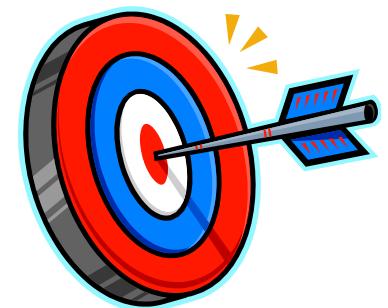
Kapitel 4  
Optional: Input- & Output-Stream

# Themenüberblick

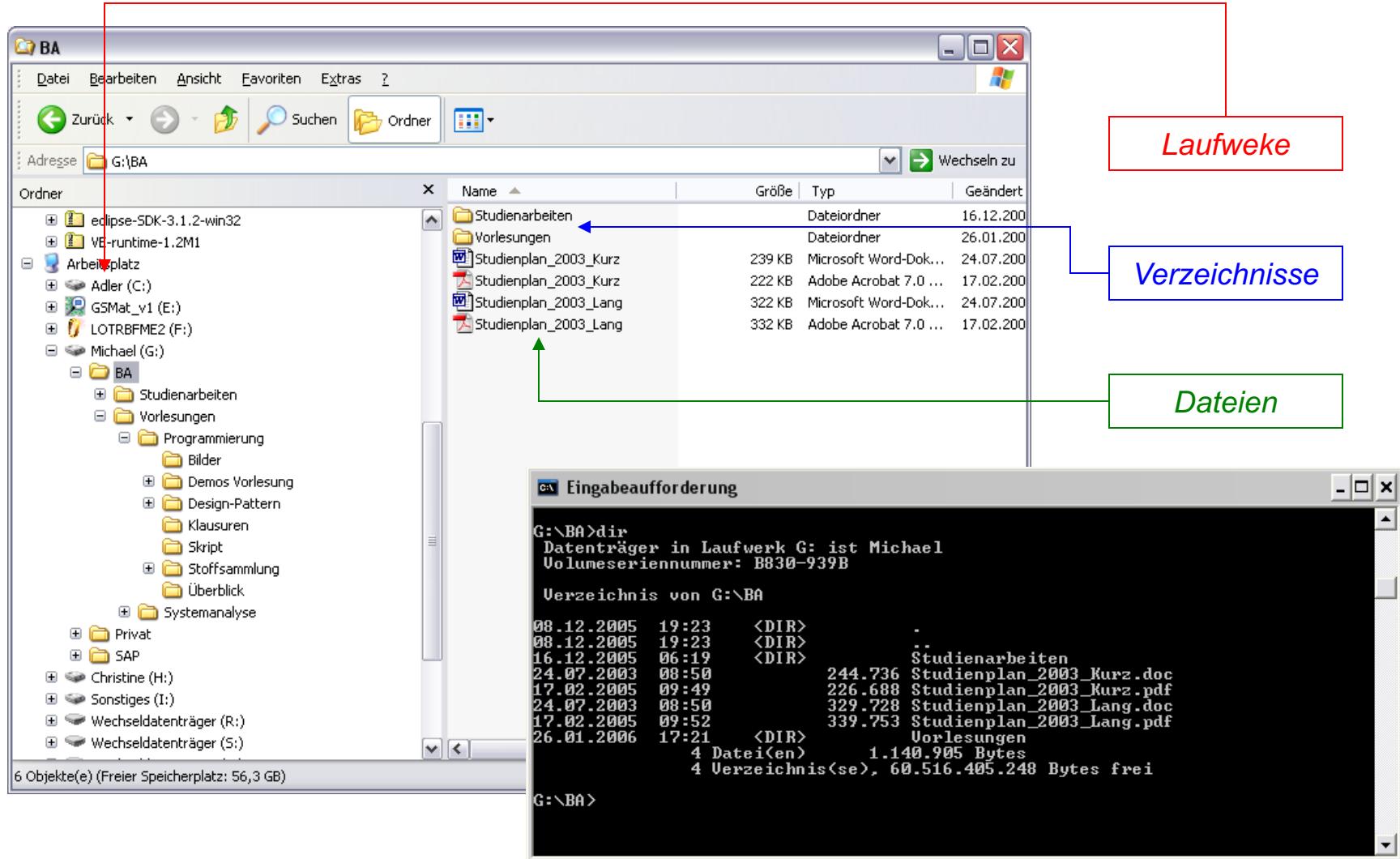


# Lernziele

- 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 Properties-Dateien anlegen, mit Werten füllen und wieder auslesen
- Sie können Anwendungen mit mehrsprachigen Texten implementieren



# Zugriff auf das Filesystem unter Windows



# 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

# Informationen über Laufwerke beschaffen

```
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"));  
        }  
    }  
}
```

Erzeugt ein Array vom Typ File, in dem alle Laufwerke aufgelistet sind

Überprüft, ob auf das Laufwerk zugegriffen werden kann und liefert ein Ergebnis vom Typ boolean

C:\ ist aktiviert  
E:\ ist aktiviert  
...  
R:\ ist deaktiviert  
S:\ 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.");
        }
    }
}
```

Vorgänger:G:\BA\Vorlesungen\Programmierung  
Pfad:G:\BA\Vorlesungen\Programmierung\Skript  
Name:Skript

- `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.");
        }
    }
}
```

```
...
Grösse der Datei: 914944 Byte
Berechtigung (r/w): true true
Zuletzt geändert: 1140260517656
```

- `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.");
        }
    }
}
```

Legt ein neues  
Verzeichnis an

Löscht ein  
bestehendes  
Verzeichnis

Benennt ein  
bestehendes  
Verzeichnis um

# Einfacher 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(); 
            }
        }
    }
}
```

Legt eine neue Datei an

Löscht eine bestehende Datei

Benennt eine bestehende Datei um

# Ein- und Ausgabeströme in Java

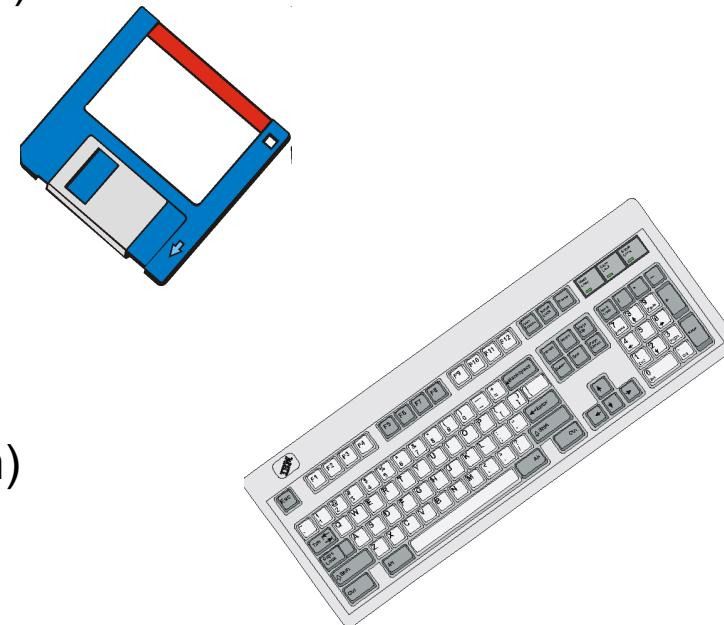
## Eingabestrom

- 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	Ausgabe des Puffers, nutzt passendes Zeilenendezeichen
ByteArrayOutputStream	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(parameter);` 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()`

# Die Methode `toString()` überschreiben

```
public class DemoToString {  
  
    private int zahl = 10;  
    public String toString() {  
        return super.toString() + " Hallo Welt " + zahl;  
    }  
    public void setZahl(int zahl) {  
        this.zahl = zahl;  
    }  
}
```

```
public class DemoToStringDemo {  
  
    public static void main(String[] args) {  
  
        DemoToString test = new DemoToString();  
        System.out.println(test);  
        test.setZahl(20);  
        System.out.println(test);  
    }  
}
```

```
prog2.demos.inout.io.DemoToString@360be0 Hallo Welt 10  
prog2.demos.inout.io.DemoToString@360be0 Hallo Welt 20
```

Es wird implizit die Methode `toString` aufgerufen

Ruft zunächst die Methode `toString()` der Superklasse `Object`

Überschreibt die Methode `toString()` der Klasse `Object`

# 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.println("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));
    }
}
```

Geben Sie einen Text ein:  
Hallo 12345  
[B@45a877  
Hallo 12345  
□□□□□□□□□□□□□□...  
]

Erzeugt eine  
EingabevARIABLE

Fängt eine  
mögliche  
IOException

Es werden maximal  
255 Zeichen von der  
Konsole gelesen

# 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));  
    }  
}
```

Geben Sie Ihren Text ein:  
12345 Hallo  
12345 Hallo

Wandelt den Byte-Stream in Zeichen-Stream um

Fängt eine mögliche IOException

Puffert die Zeicheneingabe des Input-Streams

# Fortgeschritten 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);  
            for(; text != null; text = lesePuffer.readLine()) {  
                System.out.println(text);  
            }  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Verweis auf die Quelldatei

Dies ist ein Test zum Lesen aus Dateien.

In dieser Datei können auch mehrere Zeilen enthalten sein.

Ende der Demo !!!

Lesepuffer für den FileReader

Öffnet das File-Objekt zum Lesen

# Schreiben in Textdateien

```
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();  
            }  
        }  
    }  
}
```



Verweis auf die  
Zieldatei

Nach dem  
Schreiben wird das  
File geschlossen

Öffnet das File-  
Objekt zum  
Schreiben

# 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");  
        FileInputStream leser = null;  
        FileOutputStream schreiber = null;  
        byte[] puffer = new byte[(int)quelle.length()];  
        try {  
            leser = new FileInputStream(quelle);  
            schreiber = new FileOutputStream(ziel);  
            schreiber.write(puffer, 0, leser.read(puffer));  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                leser.close();  
                schreiber.close();  
            } 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");  
        ...  
        int i = 0;  
        int puffer = 0;  
        byte[] buffer = new byte[BUF_SIZE];  
        try {  
            leser = new FileInputStream(quelle);  
            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();  
        } finally {  
            ...  
        }  
    }  
}
```

# Texte in Java-Properties-Dateien auslagern

## Vorteile der Properties

- Ziel: Auslagerung von Texten in eigener Datei
- Ablage von Schlüssel-Wertpaaren (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

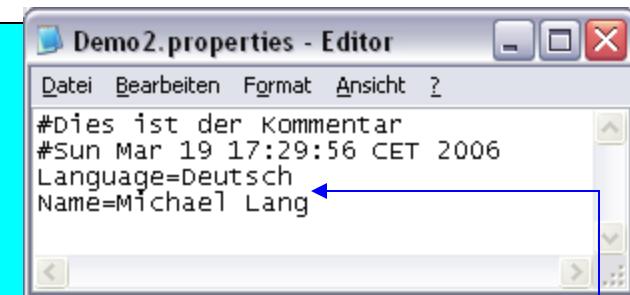
# Texte in Java-Properties-Dateien speichern

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

public class PropertiesSpeichernDemo {

    public static void main(String[] args) {

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



Erzeugt ein  
Properties-Objekt

Schreibt die  
gesetzten Properties  
in *Demo2.properties*

Setzt die Schlüssel-  
Wertpaare

# Texte aus Java-Properties-Dateien lesen

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

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

```
-- listing properties --
Language=Deutsch
Name=Michael Lang

Hallo Michael Lang
Sie bekommen die Texte in Deutsch angezeigt.
```

Lädt die Properties aus der Datei *Demo2.properties* in das Objekt *prop*

*getProperty()* liest ein konkretes Schlüssel-Wertepaar aus

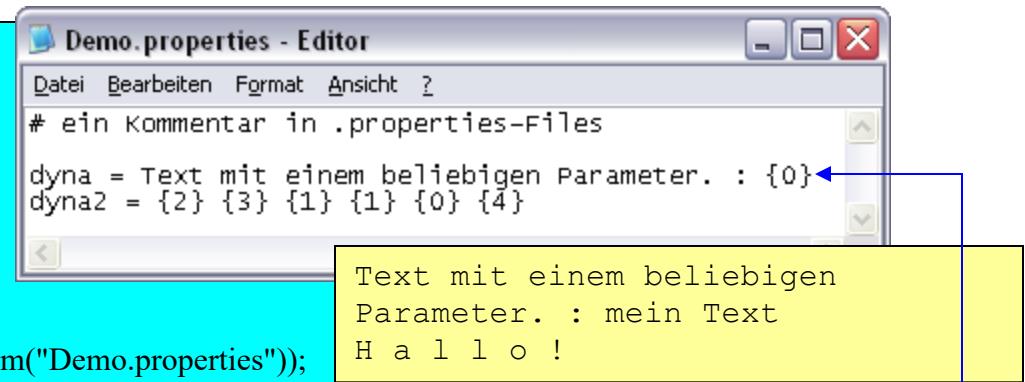
Gibt eine Liste der geladenen Properties auf der Konsole aus

# Dynamische Texte mit variablen Parametern

```
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("dyna2")); // olha
        Object[] text2 = {"o","I","H","a","!"};
        System.out.println(nachricht.format(text2));
    }
}
```

Aus dem Schlüssel-Wertpaar  
*dyna* wird eine formatierte  
Nachricht erzeugt



Es wird ein Array vom Typ  
*Object* erzeugt und mit dem  
Parameter “*meinText*“ gefüllt  
und wird an der Stelle {0}  
ausgegeben

# 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

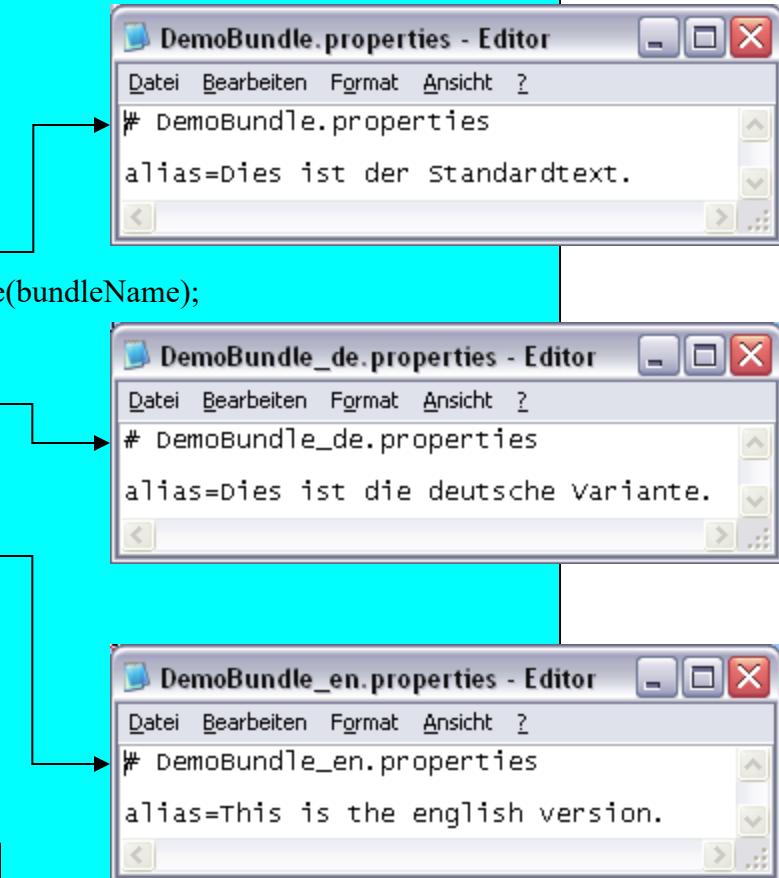
## Beispiel

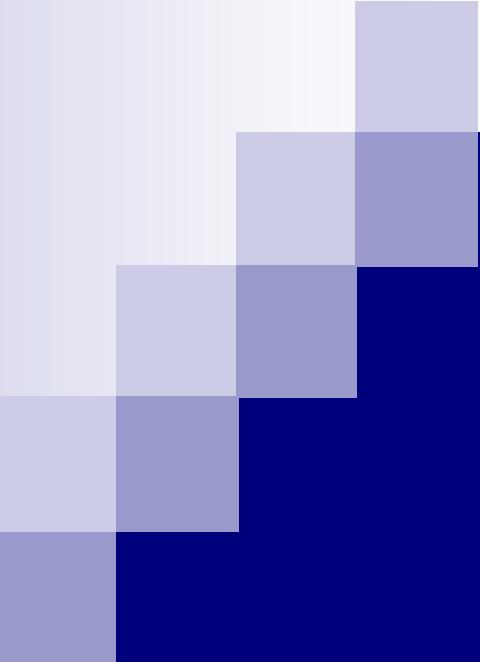
- (bundleName ist DemoBundle)
- Sprach- und Ländereinstellung ist Deutschland
- DemoBundle\_de\_DE.properties

# 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(bundle.getString("alias"));  
  
            Locale.setDefault(new Locale("de"));  
            bundle = ResourceBundle.getBundle(bundleName);  
            System.out.println(bundle.getString("alias"));  
  
            Locale.setDefault(Locale.ENGLISH);  
            bundle = ResourceBundle.getBundle(bundleName);  
            System.out.println(bundle.getString("alias"));  
  
        } catch (MissingResourceException e) {  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

Dies ist der Standardtext.  
Dies ist die deutsche Variante.  
This is the english version.

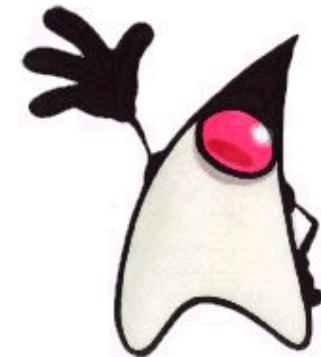
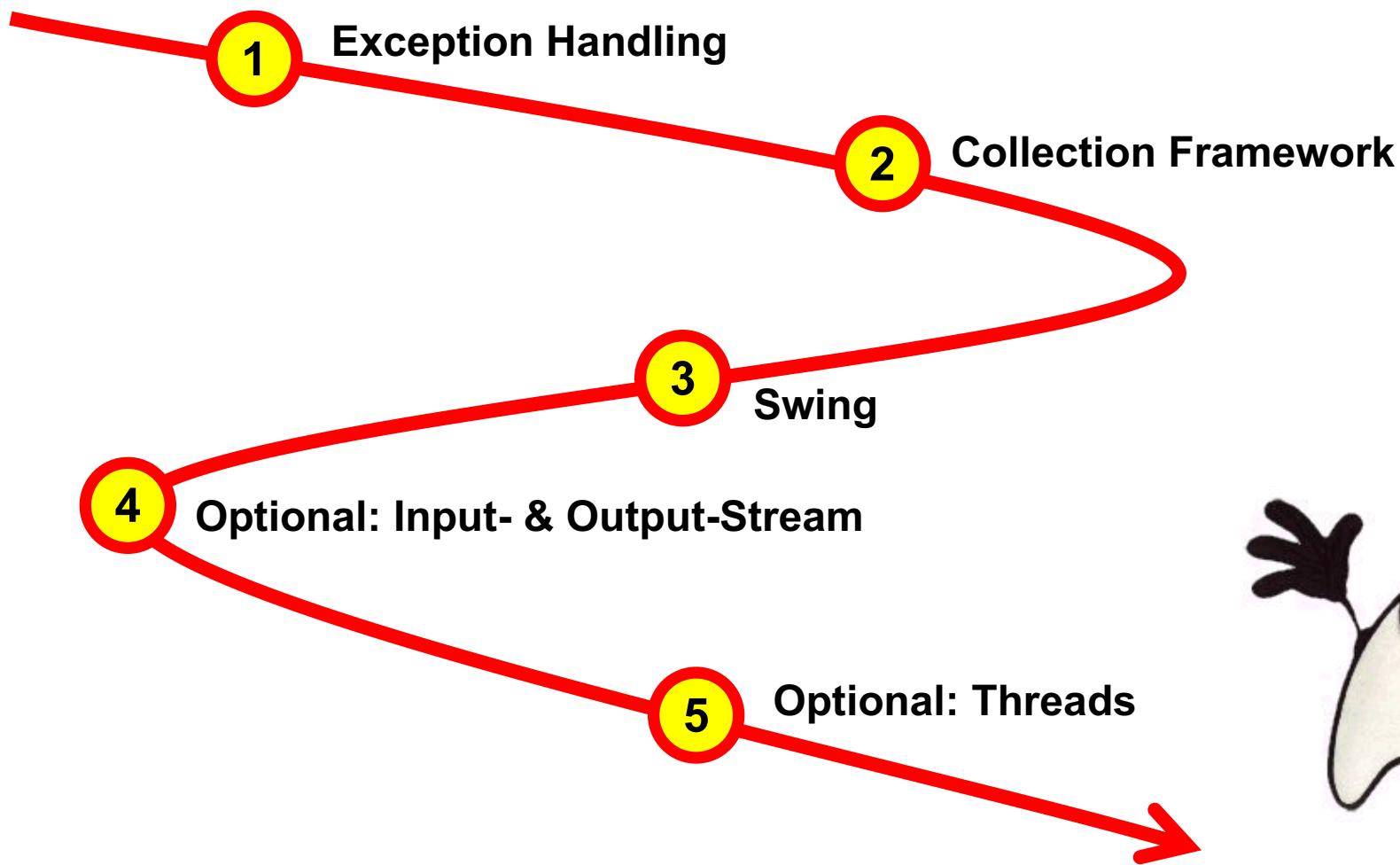




# Programmierung 2

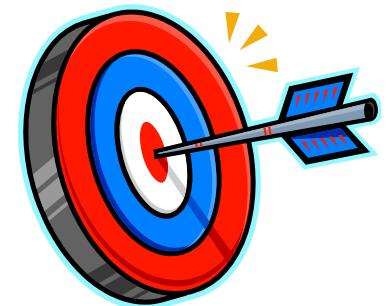
Kapitel 5  
Optional: Threads

# Themenüberblick



# Lernziele

- Sie können den Unterschied zwischen Multi-Tasking und Multi-Threading erklären
- Sie können die wesentlichen Anwendungsgebiete von Threads aufzählen
- Sie können Threads erzeugen, starten und beenden
- Sie können die unterschiedlichen Zustände eines Threads benennen und erläutern
- Sie können Threads beim Zugriff auf gemeinsame Daten synchronisieren
- Sie haben das Consumer-Producer-Problem verstanden und können es implementieren
- Sie können Threads als Dämonen kennzeichnen



# Grundlegende Begriffe zu Threads

- moderne Betriebssysteme scheinen Programme parallel auszuführen -> Multi-Tasking
- Multi-Tasking: bei Rechnern mit einem Prozessor wird zwischen Prozessen (Programme, Daten in festem Speicheradressraum, Ressourcen) in kurzen Zeitabständen umgeschaltet
- die Umschaltung wird vom Scheduler übernommen
- der Anwender nimmt eine Quasi-Parallelität der Abarbeitung wahr
- Ziel: Quasi-Parallelität nicht nur auf Betriebssystemebene, sondern innerhalb eines Programms zur Verfügung zu haben
- Java unterstützt nebenläufige Programmierung über Threads
- Threads laufen innerhalb eines Prozesses und somit innerhalb eines festen Speicherbereichs -> Zugriff auf öffentliche Attribute möglich
- falls das Betriebssystem kein Multi-Threading unterstützt, simuliert die JVM die Parallelität

# Anwendungsgebiete von Threads

- gleichzeitige Nutzung unterschiedlicher Hardware-Ressourcen
  - ◆ Hauptspeicherzugriffe
  - ◆ Prozessor
  - ◆ Dateioperationen
  - ◆ Festplatte
  - ◆ Datenbankzugriff
  - ◆ Server, Netzwerkverbindung
- Steigerung der Verarbeitungsgeschwindigkeit
- Anwendungsbeispiele für Threads
  - ◆ Öffnen eines Fensters und Öffnen bzw. Lesen einer Datei
  - ◆ Lesen von Daten aus einer Datei und Analyse bereits geladener Daten
  - ◆ Analyse neu gelesener Daten und Speichern von alten Daten in Dateien
  - ◆ typisches Beispiel: Consumer-Producer-Problematik

# Threads erzeugen und starten

## Zwei Alternativen zur Erzeugung von Threads

- Ableiten von der Klasse Thread aus dem Package java.lang
- Implementieren des Interfaces Runnable aus dem Package java.lang
- in beiden Fällen muss die Instanzenmethode run()  
überschrieben bzw. implementiert werden

## Starten von Threads

- Instanziieren einer Referenzvariable zu einer Klasse die von Thread abgeleitet ist oder das Interface Runnable implementiert
- Aufruf der Methode start() über die Referenzvariable
- die Methode start() ruft implizit die Methode run()
- nach Abarbeitung der run()-Methode wird der Thread auf Betriebssystem-Ebene von der JVM zerstört

# Weitere Aspekte im Umgang mit Threads

## Unterschiede beim Starten über start() oder run()

- kaum Unterschiede für den Anwender erkennbar
- beim Starten der Threads über start() erfolgt eine nebenläufige Abarbeitung der einzelnen Threads
- beim Starten über run() erfolgt die Abarbeitung der run()-Methoden sequentiell

## Thread-Klassen über Runnable oder Thread erzeugen

- Einschränkung ist die Einfachvererbung in Java
- Implementierung von Runnable sinnvoll, wenn von einer Super-Klasse ungleich Thread abgeleitet werden soll
- existieren keine weiteren Super-Klassen, kann von Thread abgeleitet werden

# Threads erzeugen und starten

```
public class Hase extends Thread {  
    private String name = null;  
    public Hase(String name) {  
        this.name = name;  
    }  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(name + " futtert eine Karotte!");  
        }  
    }  
}
```

```
public class Hamster implements Runnable {  
    private String name = null;  
    public Hamster(String name) {  
        this.name = name;  
    }  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(name + " futtert eine Karotte!");  
        }  
    }  
}
```

```
public class DemoThreads1 {  
    public static void main(String[] args) {  
        Hase hase = new Hase("Bugs Bunny");  
        Hamster hamster = new Hamster("Roger Rabbit");  
        Thread runThread = new Thread(hamster);  
        hase.start();  
        runThread.start();  
    }  
}
```

```
...  
Bugs Bunny futtert eine Karotte!  
Bugs Bunny futtert eine Karotte!  
Roger Rabbit futtert eine Karotte!  
Bugs Bunny futtert eine Karotte!  
Roger Rabbit futtert eine Karotte!  
Bugs Bunny futtert eine Karotte!  
...
```

# Unterschiedliche Zustände eines Threads

- es wird ein neues Thread-Objekt erzeugt, welches sich im Status new befindet
- mit der Methode start() wird der Thread in den Status ready versetzt, bis der Scheduler ihn in den Status running versetzt
- der Status running kann unterbrochen werden
  - ◆ run() wird beendet (z.B. Auslösen einer Ausnahme, return-Anweisung,etc.) und der Thread gelangt in den Status dead
  - ◆ der Thread kann das CPU-Nutzungsrecht mit der Methode yield() abgeben und ist selbst im Status ready
  - ◆ der Scheduler unterbricht den aktiven Thread und der Status des Thread ist ready
  - ◆ ein Thread kann durch Warten auf andere Threads oder durch die sleep()-Methode auf den Status blocked gesetzt werden
- der Status dead kann auch von außen erzwungen werden

# Threads von außen beenden (alte Technik)

```
public class Hase extends Thread {  
    private String name = null;  
    private boolean aktiv = true;  
  
    ...  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            if (!nochAktiv()) {  
                System.out.println("Hase ist satt!");  
                return;  
            }  
            System.out.println(name + " futtert eine Karotte!");  
        }  
    }  
    public boolean nochAktiv() {  
        return aktiv;  
    }  
    public void deaktivieren() {  
        aktiv = false;  
    }  
}
```

```
public class DemoThreads2 {  
    public static void main(String[] args) {  
        Hase hase = new Hase("Bugs Bunny");  
        Hamster hamster = new Hamster("Roger Rabbit");  
        Thread runThread = new Thread(hamster);  
        hase.start();  
        runThread.start();  
        ...  
        hase.deaktivieren();  
    }  
}
```

Bugs Bunny futtert eine Karotte!  
Bugs Bunny futtert eine Karotte!  
Roger Rabbit futtert eine Karotte!  
Hase ist satt!  
Roger Rabbit futtert eine Karotte!

Attribut, dass den Zustand des Threads beschreibt

setzt das Attribut auf false, um den Thread zu deaktivieren

sobald der Thread deaktiviert wurde, wird die Methode *run()* mit *return* beendet

# Threads über Interrupt beenden (neue Technik)

```
public class DemoInterrupt extends Thread {  
  
    public void run(){  
        while (!isInterrupted()) {  
            System.out.println("Dies ist ein Test!");  
            try {  
                Thread.sleep(20);  
            } catch (InterruptedException e) {  
                interrupt();  
                System.out.println("Abbruch");  
            }  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        DemoInterrupt test = new DemoInterrupt();  
        test.start();  
        Thread.sleep(100);  
        test.interrupt();  
    }  
}
```

Dies ist ein Test!  
Dies ist ein Test!  
Dies ist ein Test!  
Dies ist ein Test!  
Abbruch

überprüft, ob die  
Methode *interrupt()*  
das Abbruch-Flag  
gesetzt hat

setzt das  
Abbruch-Flag auf *true*

wenn der Thread beim „Schlafen“  
abgebrochen wird, wird eine  
*InterruptedException* ausgelöst und  
das Abbruch-Flag auf *false* gesetzt

# Weitere Methoden der Klasse Thread

- `sleep(long m)` oder `sleep(long m,int n)`
  - ◆ statische Methoden der Klasse Thread
  - ◆ für eine festgelegte Zeit wird der Thread vom Scheduler nicht berücksichtigt
  - ◆ die Zeitangabe erfolgt in Millisekunden m und optional in Nanosekunden n
  - ◆ checked Exception der Ausnahme `InterruptedException`
- `setPriority(int p)` und `getPriority()`
  - ◆ `setPriority(int p)` legt die Priorität des Threads fest (1 = sehr niedrig, 10 = sehr hoch)
  - ◆ die Priorität besagt, wie viel Rechenzeit ein Thread relativ zu anderen Threads vom Scheduler zugeteilt bekommt
  - ◆ mit `getPriority()` kann man die Priorität eines Threads abfragen

# Probleme beim Zugriff auf gemeinsame Daten

```
public class Buch {  
    private boolean ausgeliehen = false;  
    private String titel = "Java ist auch eine Insel";  
    public void ausleihen() {  
        if (!ausgeliehen) {  
            ...  
            ausgeliehen = true;   
            System.out.println("Buch " + titel + " wurde ausgeliehen.");  
        } else {  
            System.out.println("Buch ist schon weg!");  
        }  
    }  
}
```

```
public class Student extends Thread {  
    private Buch buch = null;  
    public Student(Buch buch) {  
        this.buch = buch;  
    }  
    public void run() {  
        buch.ausleihen();  
    }  
}
```

Buch Java ist auch eine Insel wurde ausgeliehen.  
Buch Java ist auch eine Insel wurde ausgeliehen.

```
public class DemoThreads3 {  
    public static void main(String[] args) {  
        Buch javaInsel = new Buch();  
        Student studie1 = new Student(javaInsel);  
        Student studie2 = new Student(javaInsel);  
        studie1.start();  
        studie2.start();  
    }  
}
```

# Synchronisieren von Threads

```
public class Buch {  
    private boolean ausgeliehen = false;  
    private String titel = "Java ist auch eine Insel";  
    public synchronized void ausleihen() {  
        if (!ausgeliehen) {  
            ...  
            ausgeliehen = true;  
            System.out.println("Buch " + titel + " wurde ausgeliehen.");  
        } else {  
            System.out.println("Buch ist schon weg!");  
        }  
    }  
}  
  
public class Student extends Thread {  
    private Buch buch = null;  
    public Student(Buch buch) {  
        this.buch = buch;  
    }  
    public void run() {  
        buch.ausleihen();  
    }  
}  
  
public class DemoThreads3 {  
    public static void main(String[] args) {  
        Buch javaInsel = new Buch();  
        Student studie1 = new Student(javaInsel);  
        Student studie2 = new Student(javaInsel);  
        studie1.start();  
        studie2.start();  
    }  
}
```

Buch Java ist auch eine Insel wurde ausgeliehen.  
Buch ist schon weg!

# Zeitliche Synchronisation mit wait und notify

- `wait()`, `wait(long m)` oder `wait(long m, int n)`
  - ◆ der ausführende Thread wird deaktiviert, bis ein anderer Thread die Methode `notify()` oder `notifyAll()` ausführt
  - ◆ mit `wait(long m)` oder `wait(long m, int n)` wird eine maximale Wartedauer in Milli- bzw. Nanosekunden angegeben
  - ◆ der Thread wird nach Ablauf der Wartezeit oder über die `notify()/notifyAll()-Methode` reaktiviert
- `notify()`
  - ◆ reaktiviert einen Thread, der auf das bis dahin gesperrte Objekt wartet
- `notifyAll()`
  - ◆ reaktiviert alle Threads, die auf das bis dahin gesperrte Objekt warten
  - ◆ die Threads konkurrieren um die Sperre für das Objekt
  - ◆ dabei wird ein Thread vom Betriebssystem ausgewählt

# Beispiel: Consumer-Producer-Problem

- zwei Threads arbeiten im Wechsel auf dem gleichen Datenobjekt
- der Erzeuger ändert den Wert und damit den Zustand des Datenobjektes
- der Verbraucher wartet darauf, dass der Erzeuger seine Änderung durchgeführt hat
- der Verbraucher liest den geänderten Wert aus
- der Erzeuger wartet, bis der Verbraucher seinen Teil abgearbeitet hat
- Anwendungsbeispiele für das Consumer-Producer-Problem
  - ◆ Warteschlangen mit einer bestimmten Kapazität
  - ◆ bei verteilten Systemen zur Beschaffung und Verarbeitung von Daten
  - ◆ Lese-Schreibe-Problematiken

# Beispiel: Consumer-Producer-Problem

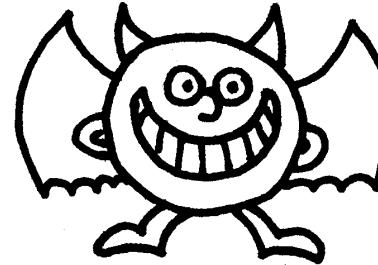
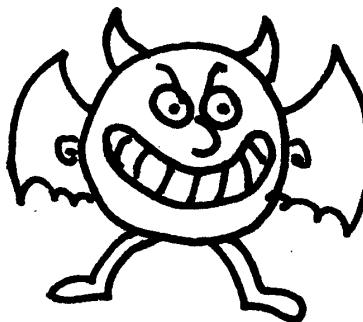
```
public class Baecker extends Thread {  
    ...  
    private synchronized void backen() throws InterruptedException {  
        while(bestand == LAGERGROESSE)  
            →wait();  
        bestand++;  
        System.out.println("Neuer Lagerbestand: " + bestand + " Keks(e)");  
        if (bestand > 0)  
            →notify();  
    }  
    public synchronized void futtern() throws InterruptedException {  
        while((bestand == 0) && (arbeitet))  
            →wait();  
        if (bestand > 0) {  
            bestand--;  
            System.out.println("Keks gefuttert!");  
            if (bestand < LAGERGROESSE)  
                →notify();  
        }  
    }  
    ...  
}
```

Neuer Lagerbestand: 1 Keks (e)  
Keks gefuttert!  
Neuer Lagerbestand: 1 Keks (e)  
Keks gefuttert!  
Neuer Lagerbestand: 1 Keks (e)  
Keks gefuttert!  
...

```
public class Monster extends Thread {  
    ...  
    public void run() {  
        while(futtern) {  
            try {  
                baecker.futtern();  
                sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
    ...  
}
```

# Dämonen - eine spezielle Art von Threads

- Dämonen sind speziell gekennzeichnete Threads
- sie werden automatisch von der Laufzeitumgebung beendet, sobald kein „normaler“ Anwendungsthread mehr läuft
- Einsatz von Dämonen bei Überwachungs- oder Serveraufgaben sinnvoll (Prozeße, Dienste, etc.)
- Kennzeichnung eines Threads mit Hilfe der Methode `setDaemon(true)`, die vor dem Start des Threads ausgeführt werden muss
- ist ein Thread als Dämon gekennzeichnet, kann er nicht mehr zurückverwandelt werden



# Beispiel für die Implementierung eines Dämons

```
public class MeinDaemon extends Thread {  
    private boolean auftrag = false;  
    private String nachricht = null;  
  
    public MeinDaemon(){  
        setDaemon(true);  
    }  
    public void run() {  
        while (true) {  
            if (auftrag) {  
                System.out.println(nachricht);  
                auftrag = false;  
            }  
        }  
    }  
    public void setNachricht(String message) {  
        nachricht = message;  
        auftrag = true;  
    }  
}
```

```
Dies ist Nachricht Nr. 0  
Dies ist Nachricht Nr. 1  
Dies ist Nachricht Nr. 2  
Dies ist Nachricht Nr. 3  
Dies ist Nachricht Nr. 4  
...
```

```
public class DemoDaemon{
```

```
    public static void main(String[] args) throws InterruptedException {  
        MeinDaemon daemon = new MeinDaemon();  
        daemon.start();  
  
        for (int i = 0; i < 10; i++) {  
            daemon.setNachricht("Dies ist Nachricht Nr. " + i);  
            Thread.sleep(10);  
        }  
    }
```

# Literaturverzeichnis

ULLENBOOM, Christian: Java ist auch eine Insel, Galileo Press, 3. Auflage 2003,  
ISBN 3-89842-365-4

ULLENBOOM, Christian: Java ist auch eine Insel, Galileo Press, 5. aktualisierte und erweiterte  
Auflage 2006, ISBN 3-89842-747-1

KRÜGER, Guido: Handbuch der Java-Programmierung, Addison-Wesley, 4. veränderte Auflage 2006,  
ISBN 3-8273-2361-4

MISCH, Jens-Peter: Java 4 U Programmentwicklung mit Java, Bildungsverlag EINS, 1. Auflage 2003,  
ISBN 3-427-01144-5