Kapitel 5: Objekte verwalten

Generics

```
public class Safe {

    private Object secretObject;

    public Safe(Object secret) {
        secretObject = secret;
    }

    public Object getSecret(String password) {
        if (password.equals("sesam"))
            return secretObject;
        else
            return null;
    }
}
```

Speichert ein Objekt eines beliebigen Typs.

Anhand des korrekten Passworts kann das Objekt wieder ausgelesen werden.

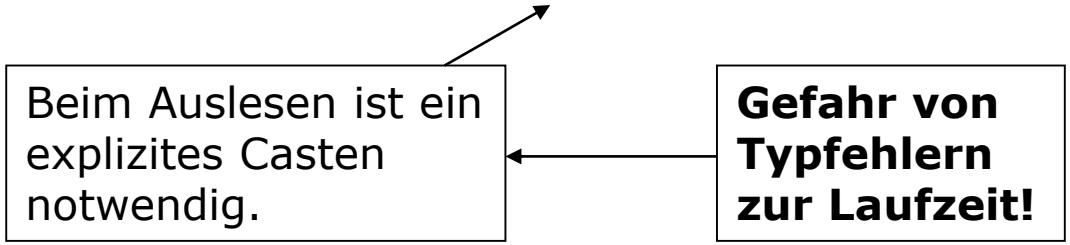
Anwendung

- Beispielsweise wird ein Rechteck übergeben…

```
Rechteck in = new Rechteck(10, 10, 50, 80);
Safe safe = new Safe(in);
```

- Und schließlich wieder ausgelesen…

```
Rechteck out = (Rechteck)safe.getSecret("sesam");
```



- Typsichere Alternativen:
  - Spezielle Klasse `Safe` je Typ
  - **Generics**

Generics

- Parametrisierte Klasse:

```
public class Safe<T> {

    private T secretObject;

    public Safe(T secret) {
        secretObject = secret;
    }
    public T getSecret(String password) {
        if (password.equals("sesam")) return secretObject;
        else                          return null;
    }
}
```

`Safe<T>` ist ein generischer Typ bzw. ein parametrisierter Typ

`T` ist ein Typparameter

- Anwendung:

```
Rechteck in = new Rechteck(10, 10, 50, 80);
Safe<Rechteck> safe = new Safe<Rechteck>(in);
…
Rechteck out = safe.getSecret("sesam");
```

Konkretisierung von `T`, das Typargument

Kein Cast-Operator notwendig

Generics allgemein

- Generics bieten die Möglichkeit **typsichere** Container-Klassen zu programmieren.
- **Container-Klassen** speichern Objekte beliebigen Typs und sind dabei nicht an deren konkreten Typ interessiert, d.h. sie speichern Objekte als ganzes und wenden nicht z.B. deren Methoden an.
- Elementare Typen sind als Typargumente nicht möglich, sondern **nur Referenztypen**!
- **Collections** bzw. **Sammlungen** sind spezielle Container-Klassen, die **mehrere** Objekte **gleichen** Typs verwalten (z.B. Listen und Mengen).

Die Klasse `Gruppe`

```
public class Gruppe extends Flaeche {

   private ArrayList<Flaeche> flaechenListe;

   public Gruppe(int x, int y) {
      super(x, y);
      flaechenListe = new ArrayList<Flaeche>();
   }
   public void add(Flaeche f) {
      flaechenListe.add(f);
   }
   public void remove(Flaeche f) {
      flaechenListe.remove(f);
   }
   public void paint(Graphics g) {
      // todo, Flaechen der Gruppe zeichnen
   }
}
```

Speichert eine Liste von `Flaeche`-Objekten

`Flaeche` der Liste hinzufügen

`Flaeche` aus der Liste löschen

Kommt noch…

Collections in Java

Grundtypen:

- Listen
  - Basisinterface: `List<E>`
  - Beispiel: `ArrayList<E>`
  - Geordnet
  - Duplikate erlaubt
- Mengen
  - Basisinterface: `Set<E>`
  - Beispiel: `HashSet<E>`
  - I.d.R. ungeordnet
  - Duplikate nicht erlaubt
- Zuordnungen
  - Basisinterface: `Map<K,V>`
  - Beispiel: `HashMap<K,V>`
  - Schlüssel-Werte-Paare
  - Schlüssel sind wie Mengen

Die Klasse `ArrayList<E>`

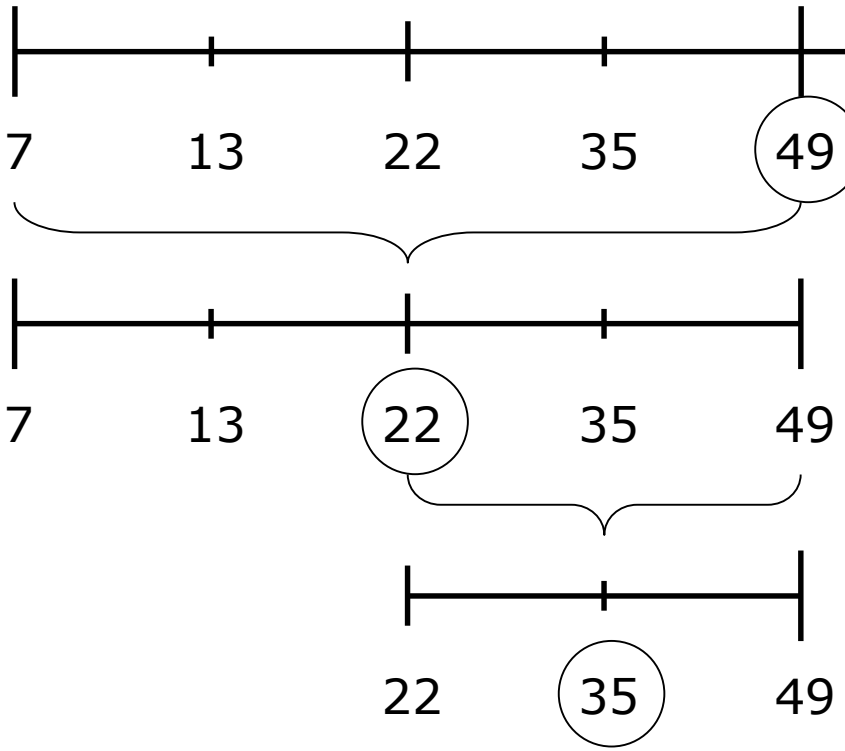
`boolean add(E e)`	Hängt das Element `e` am Ende der Liste an.
`void add(int index, E e)`	Fügt das Element `e` an einer bestimmten Position ein
`void clear()`	Entfernt alle Element aus der Liste
`Object clone()`	Gibt eine Kopie der Instanz zurück
`boolean contains(Object o)`	Prüft, ob ein bestimmtes Element in der Liste enthalten ist
`E get(int index)`	Gibt das Element an einer bestimmten Position zurück
`int indexOf(Object o)`	Gibt den Index eines Elements zurück. Falls `o` mehrmals in der Liste enthalten ist, wird der erste Index zurückgegeben. Fall `o` nicht in der Liste enthalten ist, wird -1 zurückgegeben
`E remove(int index)`	Entfernt ein Element an einer bestimmten Position
`boolean remove(Object o)`	Entfernt ein gegebenes Element aus der Liste. Falls `o` mehrmals in der Liste enthalten ist, wird nur die erste Instanz gelöscht. Falls `o` nicht in der Liste enthalten ist, wird `false` zurückgegeben, sonst `true`
`E set(int index, E e)`	Ersetzt ein Element an einer bestimmten Position durch ein neues Element `e`. Gibt das alte Element zurück
`int size()`	Gibt die Anzahl der Listenelemente zurück

Zuordnungen

- Anwendung Telefonbuch:
  - Name -> Telefonnummer
  - (Key -> Value)
- Die Suche im Telefonbuch geht schnell, weil die Namen sortiert sind.
- Automatisiert ist die Suche in *sortierten* Listen z.B. mittels **Bisektion** möglich.

Bisektions- bzw. Binäre Suche

- Gesucht: 35



- Algorithmus
  - Mittleren Wert lesen
  - Wenn Treffer: Abbruch
  - Sonst: Liste halbieren
  - Für relevante Hälfte den Algorithmus wiederholen
- Maximal notw. Zugriffe: $\log_2(n)$, also z.B. für n =
  - 1000: 10 Zugriffe
  - 1 Mio.: 20 Zugriffe
  - 1 Mrd.: 30 Zugriffe

Hashtabelle

- … stellt eine Map dar.
- … wendet **keine** Bisektion an.
- … benötigt keine sortierte Liste.
- Der Zugriff ist meist schneller als bei der binären Suche!
- Trick: die Position des Schlüssels wird *berechnet*!
- Die Klasse `HashMap` implementiert eine Hashtabelle.

Anwendung

```
public static void main(String[] args) {

    HashMap<String, Kreis> map =
                        new HashMap<String, Kreis>();

    map.put("A", new Kreis(0, 0, 10));
    map.put("B", new Kreis(0, 0, 20));
    map.put("C", new Kreis(0, 0, 30));

    System.out.println("B mit dem Radius " +
                        map.get("B").getRadius() +
                        " wird jetzt gleich gelöscht!");
    map.remove("B");
    System.out.println(map.get("B"));
}
```

Gibt `null` aus.

Funktionsweise von `HashMap<K,V>`

- Die Key-Klasse implementiert einen Methode `int hashCode()`.
- Für *gleiche* Instanzen muss `hashCode()` denselben Wert zurückgeben.
- Unterschiedliche Instanzen haben i.d.R. unterschiedliche Hashcodes. Das kann aber nicht immer gewährleistet werden. Sie können also auch gleich sein.
- Ob zwei Instanzen *gleich* sind, bestimmt die `boolean equals()`-Methode. Auch sie wird von der Key-Klasse implementiert.

Funktionsweise von `HashMap<K,V>`

- `HashMap` hat intern einen Array vom Typ `Entry`, mit den Attributen `key`, `value`, `next` und `hash`.
- Beim Setzen eines Eintrags wird der Hashcode des Keys ausgelesen und dann so modifiziert, dass sein Wert unter dem der Länge des internen Arrays liegt.
- Dadurch ist der Index im Array bestimmt.
- Falls dort bereits ein Eintrag gespeichert ist, wird er durch den neuen Eintrag ersetzt und der neue verweist dann auf den alten Eintrag (mit `next`, lineare Liste).
- Falls sich dort bereits ein *gleicher* Eintrag befindet (durch `equals()` bestimmt), wird der alte Eintrag überschrieben.

Funktionsweise von `HashMap<K,V>`

- Beim Auslesen wird der Hashcode analog berechnet.
- Bei Überschneidungen muss die lineare Liste durchlaufen werden, bis Gleichheit mit dem gesuchten Key herrscht.
- Um Überschneidungen möglichst zu verhindern, wird der interne Array nur bis zu einem bestimmten Prozentsatz gefüllt, dem sog. `LoadFactor`. Dieser ist standardmäßig 75%. Ist er überschritten, wird der Array in einen doppelt so großer Array umkopiert.
- `hashCode()` und `equals()` besitzen eine Standardimplementierung in `Object`. Der `hashCode()` ist dabei die Adresse der Instanz, `equals()` vergleicht die Adressen.
- Beachten Sie: gleiche Instanzen müssen dieselben Hashcodes erzeugen. I.d.R. berücksichtigen `equals()` und `hashCode()` dieselben Attribute des Keys.

Wichtige Methoden von `HashMap<K,V>`

`HashMap(`
`  int initialCapacity,`
`  float loadFactor)`
	Erzeugt eine `HashMap` mit einer definierten Ausgangskapazität und einem definierten Ladefaktor

`HashMap()`	Erzeugt eine `HashMap` mit einer Kapazität von 16 und einem Ladefaktor von 0.75
`void clear()`	Entfernt alle Einträge aus der Map
`Object clone()`	Gibt eine Kopie der Instanz zurück
`boolean containsKey(` `  Object key)`	Prüft, ob ein bestimmter Schlüssel in der Map enthalten ist
`V get(Object key)`	Gibt den Wert zu einem bestimmten Schlüssel zurück
`V put(K key, V value)`	Setzt ein Schlüssel-Wert-Paar. Gibt den bisherigen Wert zu diesem Schlüssel zurück. Falls der Schlüssel noch nicht gesetzt war, wird `null` zurückgegeben
`V remove(Object key)`	Entfernt ein Element mit einem bestimmten Schlüssel
`int size()`	Gibt die Anzahl der Einträge zurück

Die Datenstrukturen Queue und Stack

Queue (Warteschlange)

- Verarbeitung nach FIFO (First-In-First-Out)
- In Java:
  `java.util.Queue<E>`
- Wichtige Methoden:
  `boolean offer(E e)`
  `E poll()`
  `E peek()`

Stack (Stapel, Keller)

- Verarbeitung nach LIFO (Last-In-First-Out)
- In Java:
  `java.util.Stack<E>`
- Wichtige Methoden:
  `E push(E item)`
  `E pop()`
  `E peek()`

Erweiterte for-Schleife

- … zum Durchlaufen von Collections
- Die fehlende `paint()`-Methode von `Gruppe`:

```
public void paint(Graphics g) {
    for (Flaeche f : flaechenListe)
        f.paint(g);
}
```

- Aber die Elemente sollten relativ zur Gruppe positioniert werden!

Der zweite Versuch…

- Mit `translate()` lässt sich der Ursprung verschieben.

- ```
  public void paint(Graphics g) {

      g.translate(getX(), getY());

      for (Flaeche f : flaechenListe)
          f.paint(g);

      g.translate(-getX(), -getY());
  }
  ```
  ```

Voraussetzungen für Collections zur Nutzung in der erweiterten for-Schleife

- (Sorry, für die lange Überschrift!)
- Die Collection muss das Interface `Iterable<T>` implementieren:

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

- Der erzeugte Iterator muss das Interface `Iterator<T>` implementieren:

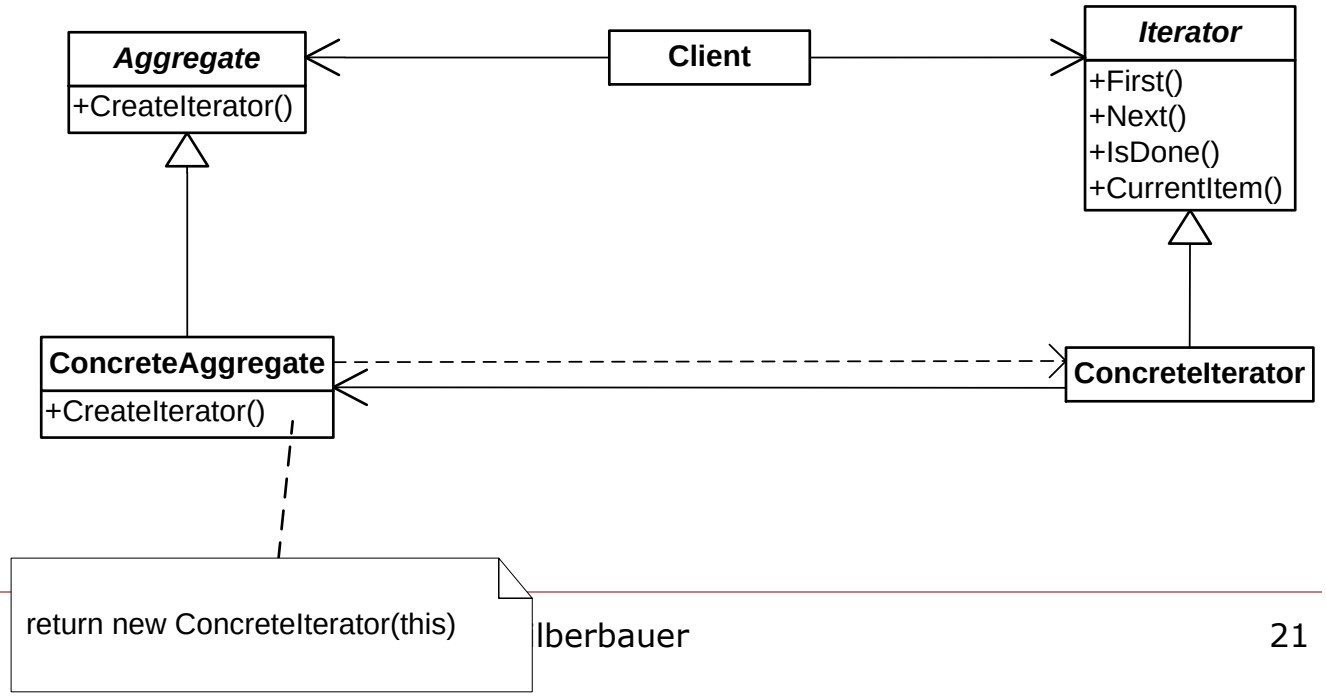
```
public interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```

[bs] Iterator-Pattern

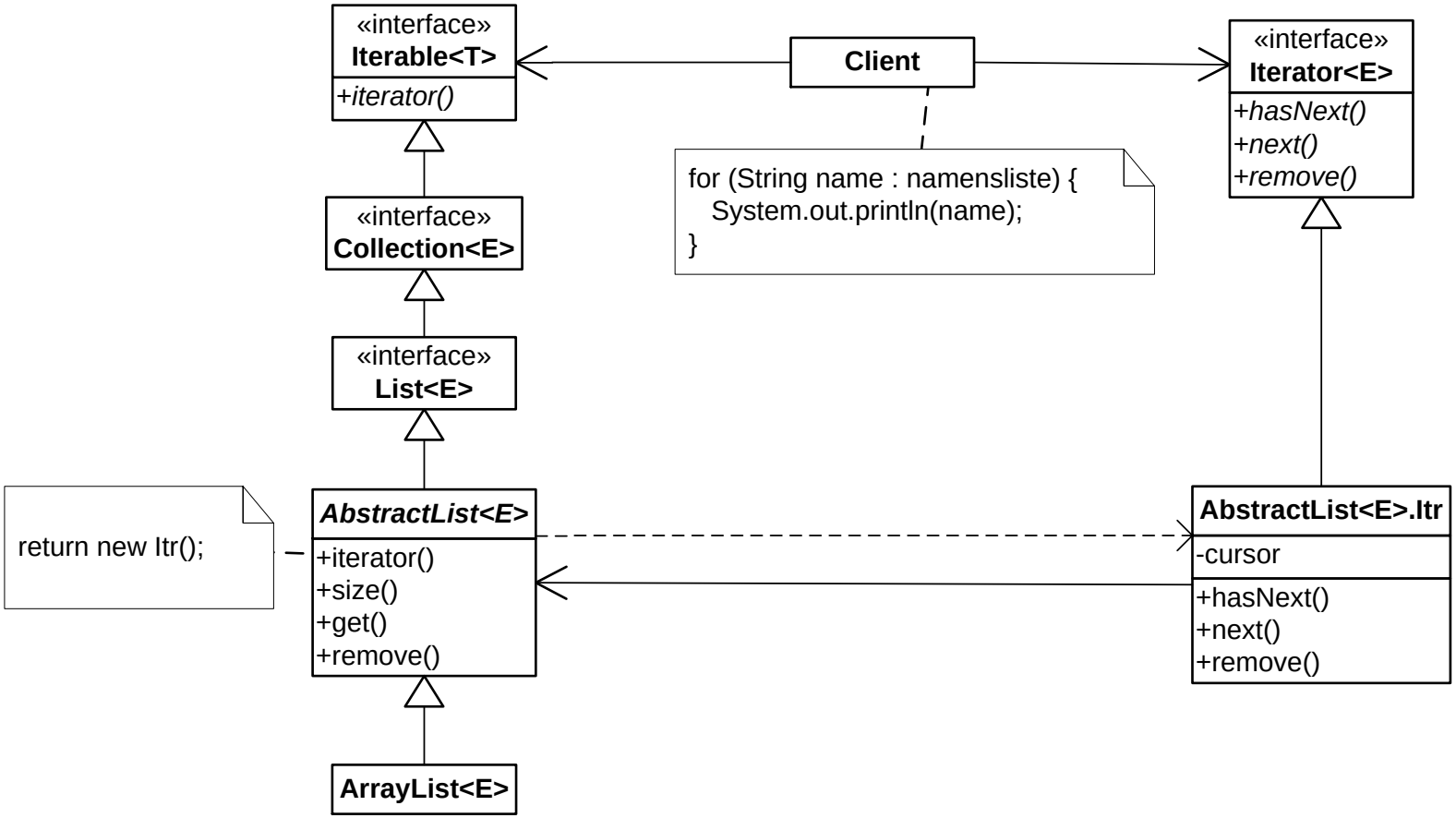
- Zweck:

  Bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Objekt sequentiell zuzugreifen, ohne die zu Grunde liegende Implementierung zu offenbaren.

- Struktur:



[bs] Beispiel: Java-`ArrayList<E>`



Iterieren von Hashmaps

```
public static void main(String[] args) {

   HashMap<String, Kreis> map = new HashMap<String, Kreis>();

   map.put("A", new Kreis(0, 0, 10));
   map.put("B", new Kreis(0, 0, 20));
   map.put("C", new Kreis(0, 0, 30));

   for (String key : map.keySet()) {
      System.out.println(key + ": " + map.get(key));
   }

   for (Kreis kreis : map.values()) {
      System.out.println(kreis);
   }
}
```

Über Keys…

Über Values…

...

- Die (bisherige) `add()`-Methode in `Gruppe`:

```
public void add(Flaeche flaeche) {
    flaechenListe.add(f);
}
```

- Es kann immer nur eine `Flaeche` hinzugefügt werden…

- Die neue `add()`-Methode:

```
public void add(Flaeche... flaechen) {
    for (Flaeche f : flaechen)
        flaechenListe.add(f);
}
```

...

- Der …-Operator erlaubt eine variable Anzahl von Methodenargumenten.

- Anwendung:

```
Rechteck r = new Rechteck(0,0,190,100);
Kreis k1 = new Kreis(10,10,40);
Kreis k2 = new Kreis(10,100,40);
Gruppe g = new Gruppe(50,50);
g.add(r, k1, k2);
```

- Der entsprechende Parameter verhält sich wie ein Array.

- Er kann mit anderen Parametern kombiniert werden, muss dann aber der letzte in der Parameterliste sein.

```
public class Gruppe extends Flaeche {

   private ArrayList<Flaeche> flaechenListe;

   public Gruppe(int x, int y) {
      super (x, y);
      flaechenListe = new ArrayList<Flaeche>();
   }
   public void add(Flaeche... flaechen) {
      for (Flaeche f : flaechen)
         flaechenListe.add(f);
   }
   public void remove(Flaeche f) {
      flaechenListe.remove(f);
   }
   public void paint(Graphics g) {
      g.translate(getX(), getY());
      for (Flaeche f : flaechenListe) f.paint(g);
      g.translate(-getX(), -getY());
   }
}
```

Eine „Zeichnung“

```
public static void main(String[] args) {

   Zeichenprogramm z = new Zeichenprogramm();

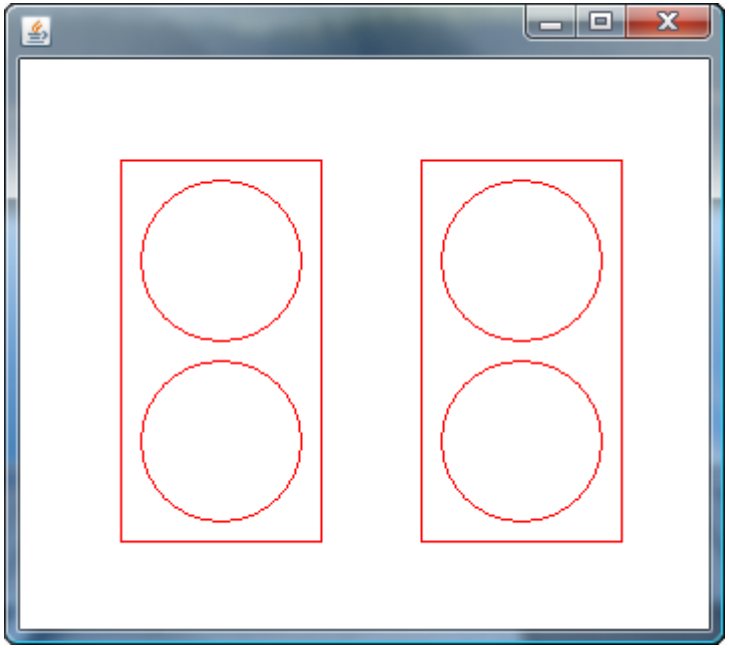
   Gruppe root = new Gruppe(0,0);

   Rechteck r = new Rechteck(0,0,100,190);
   Kreis k1 = new Kreis(10,10,40);
   Kreis k2 = new Kreis(10,100,40);

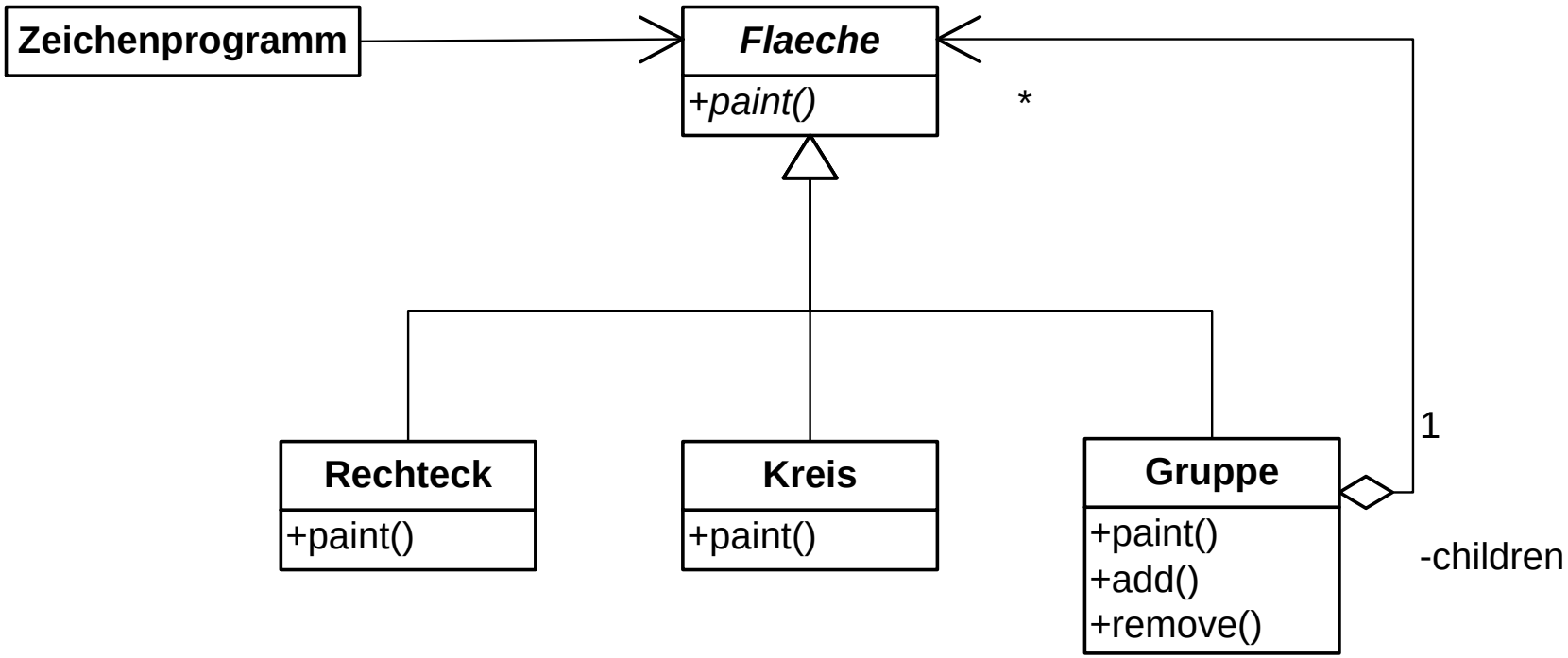
   Gruppe g1 = new Gruppe(50,50);
   g1.add(r, k1, k2);

   Gruppe g2 = new Gruppe(200,50);
   g2.add(r, k1, k2);
   root.add (g1, g2);
   z.setFlaeche(root);
}
```

Die Ausgabe



Das Klassendiagramm



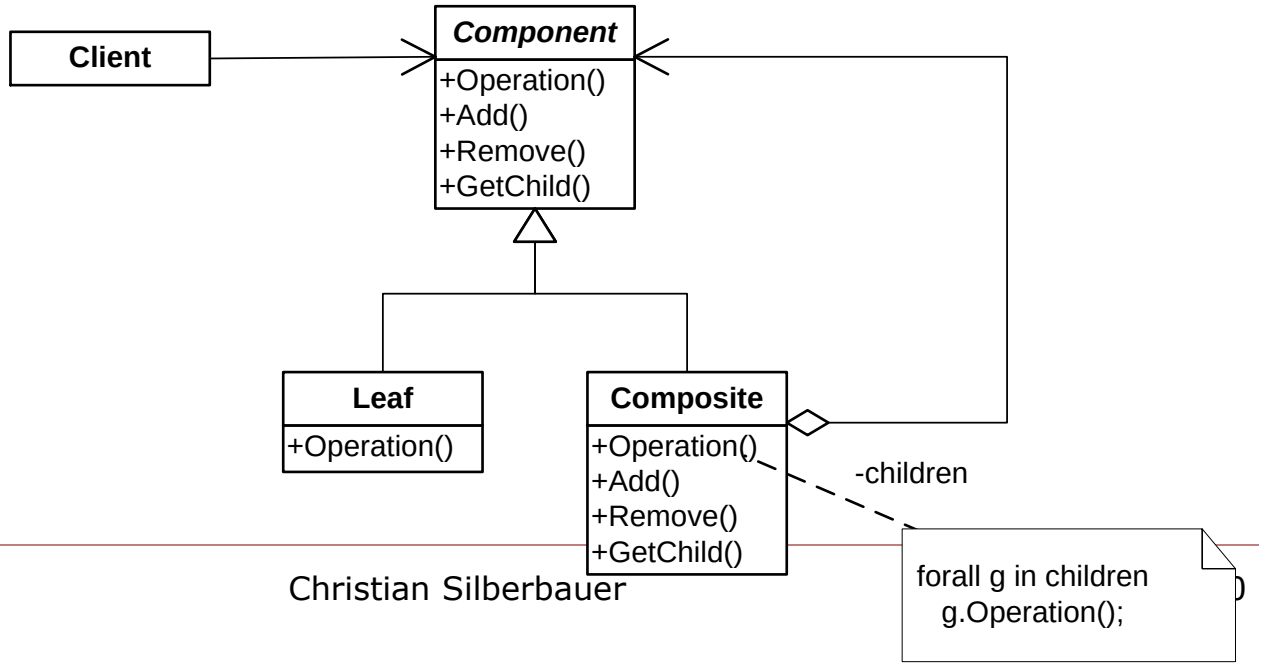
➔ Wendet das **Composite-Pattern** an

[bs] Composite-Pattern

- Zweck:

Ermöglicht es, Objekte zu einer Baumstruktur zusammenzusetzen und Teil/Ganzes-Hierarchien auszudrücken. Composite erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.

- Struktur:



Enumerationen

- Zurück zur KundenVerwaltung v1:

```
public class KundeBearbeitenDialog extends JDialog {

   public KundeBearbeitenDialog() {
      setTitle("Kunde bearbeiten");
      setSize(400, 300);
      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
      setVisible(true);
   }
   …
}
```

- `setDefaultCloseOperation()` ist wie folgt deklariert:

```
public void setDefaultCloseOperation(int operation)
```

- Wo liegt das Problem?

Enumerationen

- Hier die Deklaration der relevanten Konstanten:

```
public interface WindowConstants
{
    public static final int DO_NOTHING_ON_CLOSE = 0;
    public static final int HIDE_ON_CLOSE = 1;
    public static final int DISPOSE_ON_CLOSE = 2;
    public static final int EXIT_ON_CLOSE = 3;
}
```

- Was passiert bei `EXIT_ON_CLOSE`?

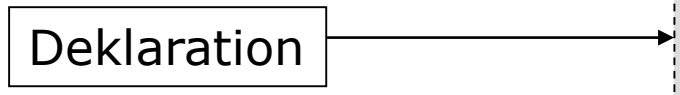
  ➔ Exception! Nicht bei `JDialog`, nur bei `JFrame` erlaubt!
- Was passiert bei `JLabel.LEFT`?

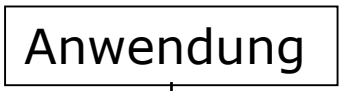
  ➔ Wird akzeptiert, sollte aber nicht…

Enumerationen

- Problem der `int`-Konstanten ist die fehlende Typsicherheit.
- Fehler können wenn überhaupt erst zur Laufzeit festgestellt werden.
- Lösung: **Enumerationen**!
- (Die gibt es aber erst seit Java 1.5, `WindowConstants` gibt es schon länger…)

Beispiel






```
enum DialogDefaultCloseOperation {
    DoNothingOnClose,
    HideOnClose,
    DisposeOnClose
}

enum FrameDefaultCloseOperation {
    DoNothingOnClose,
    HideOnClose,
    DisposeOnClose,
    ExitOnClose
}

enum HorizontalAlignment {
    Left, Center, Right
}
```

```
public class JDialog … {
    …
    private DialogDefaultCloseOperation defaultCloseOperation;

    public void setDefaultCloseOperation(DialogDefaultCloseOperation operation) {
        defaultCloseOperation = operation;
    }
}
```

```
JDialog d = new JDialog();
d.setDefaultCloseOperation(DialogDefaultCloseOperation.DisposeOnClose); // OK
d.setDefaultCloseOperation(FrameDefaultCloseOperation.DisposeOnClose);  // Fehler
d.setDefaultCloseOperation(HorizontalAlignment.Left);                   // Fehler
```

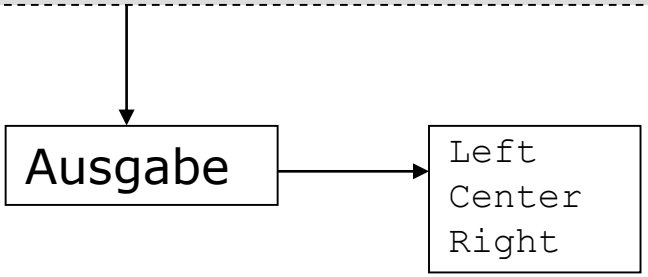
Details

- Enum-Typen sind von der Klasse `java.lang.Enum<E>` abgeleitet.
- Enum-Werten sind Ordinalzahlen zugewiesen, aufsteigend, beginnend mit dem Wert 0.
- Enum-Typen können in `switch`-Anweisungen verwendet werden.
- Enum-Typen sind Klassen, damit können sie auch eigene Methoden implementieren.

Anwendung

```
HorizontalAlignment ha = HorizontalAlignment.valueOf("Left");
                                    // HorizontalAlignment.Left
int x = HorizontalAlignment.Left.ordinal();      // 0
String s = HorizontalAlignment.Left.toString(); // "Left"
```

```
for (HorizontalAlignment ha : HorizontalAlignment.values()) {
   System.out.println(ha);
}
```



Exceptions

- Zur Behandlung von Ausnahmesituationen im Programmablauf
- Beispiel: Der Zugriff auf eine Datei klappt nicht.
- Alternative Prüfung mit if-Anweisungen bietet keine elegante Syntax.

Exceptions: `try-catch`

```
public static void dateiVerarbeiten()

    try {
        // Öffne Datei

        while(/* Dateiende nicht erreicht */) {

            // Lies Zeile aus Datei
        }

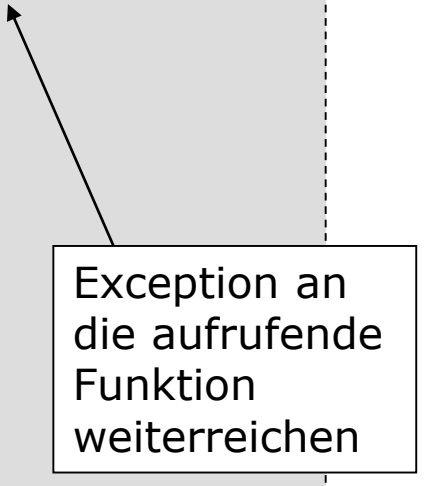
        // Schließe Datei
    }
    catch (Exception e) {
        …
    }
}
```

Exception abfangen und Fehlerbehandlung durchführen

Exceptions: `throws`

```
public static void dateiVerarbeiten() throws Exception {
    // Öffne Datei
    while (/* Dateiende nicht erreicht */) {
        // Lies Zeile aus Datei
    }
    // Schließe Datei
}

public static void main(String[] args) {
    try {
        dateiVerarbeiten();
    }
    catch (Exception e) {
        …
    }
}
```



Exceptions: mehrere `catch`-Klauseln

```
public static void dateiVerarbeiten()

    try {
        …
    }
    catch (ArrayIndexOutOfBoundsException e) {
        …
    }
    catch (IOException e) {
        …
    }
    catch (Exception e) {
        …
    }
}
```

Reihenfolge der `catch`-Klauseln wird berücksichtigt; Allgemeinere (in der Vererbunghierarchie weiter oben befindende) Exceptions daher stets nach den spezielleren Exceptions platzieren

Exceptions: `finally`

```
public static void main(String[] args) {
    try {
        dateiVerarbeiten();
    }
    catch (Exception e) {
        System.out.println(e);
    }
    finally {
      …
    }
}
```

Wird in jedem Fall durchlaufen

Runtime-Exceptions

- Typischerweise systemnahe Exceptions
- Sind von der Klasse `RuntimeException` abgeleitet
- Keine `throws`-Klausel zum Weiterreichen notwendig
- Beispiele:
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `NumberFormatException`

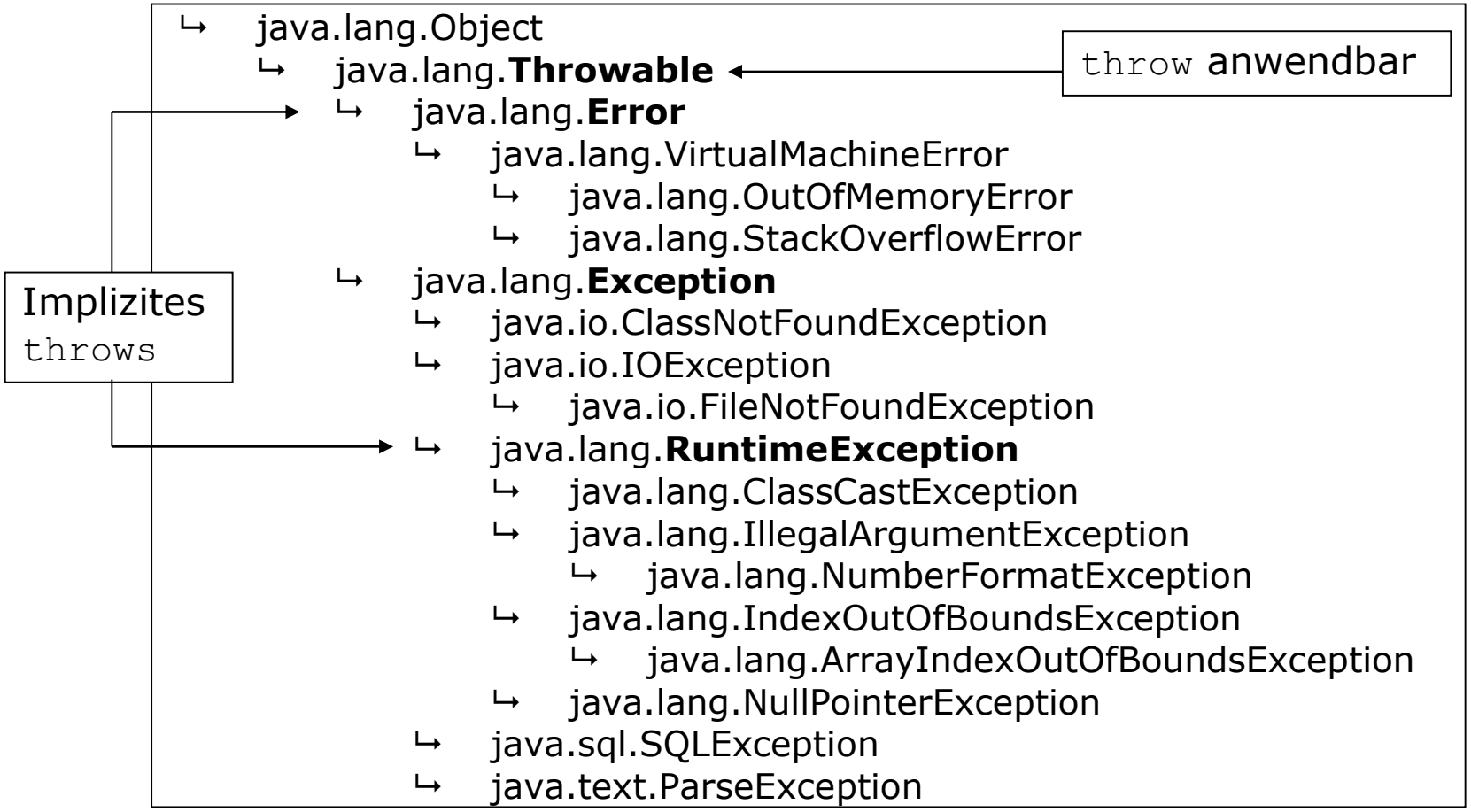
Errors

- Typischerweise systemkritische Throwables, die i.d.R. nicht abgefangen werden sollten
- Sind von der Klasse `Error` abgeleitet
- Keine `throws`-Klausel zum Weiterreichen notwendig
- Beispiele:
  - `OutOfMemoryError`
  - `StackOverflowError`

`Throwable`

- Nur eine Instanz von `Throwable` oder eine ihrer Unterklassen kann mit `throw` *geworfen* werden.
- Ist Oberklasse aller Exceptions und Errors
- Instanzen beinhalten:
  - Den Funktionsstack zum Zeitpunkt der Instanziierung
  - Optional einen Beschreibungstext
  - Optional eine Ursache (*Cause*); das ist wiederum ein `Throwable` welches weitergereicht wurde (per `throws`) und das `Throwable` ausgelöst hat. So kann eine Kette von `Throwable`s entstehen.

Klassenhierarchie



Wrapper-Klassen für Elementare Typen

Elementarer Typ (plus void)	Wrapper-Klasse
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`double`	`Double`
`float`	`Float`
`boolean`	`Boolean`
`char`	`Character`
`void`	`Void`

Wrapper-Klassen für Elementare Typen

- Wrapper-Klassen sind Referenztypen, ihre Instanzen sind Objekte.
- Im Gegensatz dazu sind Elementare Typen Wertetypen.
- Wrapper-Klassen besitzen ein Attribut ihres Elementaren Typs.
- Sie bieten die Konvertierung von und zu ihren elementaren Pendants, erlauben Stringkonvertierungen und definieren Hilfreiche Konstanten im Umgang mit den Elementaren Typen (z.B. `Integer.MAX_VALUE`)

Beispiel: `Integer`

`static int MAX_VALUE`	Maximaler Wert eines `int` (2^{31}-1)
`static int MIN_VALUE`	Minimaler Wert eines `int` (-2^{31})
`Integer(int value)`	Instanziierung anhand eines `int`-Werts
`Integer(String s)`	Instanziierung anhand eines `String`-Objekts
`int intValue()`	Gibt den internen `int`-Wert zurück
`static int parseInt(String s)`	Konvertiert einen `String` in ein `int`
`String toString()`	Gibt die `String`-Repräsentation zurück

Beispiel: `Integer`

- Umwandlung eines `String`s in einen `int`:

```
String s = "-12345";

try {
   int i = Integer.parseInt(s);
   System.out.println(i);
}
catch (NumberFormatException e) {
   …
}
```

Eine `RuntimeException`!

Autoboxing

- Bereits bekannt: Elementare Typen sind nicht als Typargumente parametrisierter Klassen erlaubt.
- In diesem Fall ist die Verwendung von Wrapper-Klassen notwendig.

Beispiel

```
public class ZahlenListe {
    private List<Integer> zahlenListe;

    public ZahlenListe() {
        zahlenListe = new ArrayList<Integer>();
    }

    public void addZahl(int zahl) {
        Integer h = new Integer(zahl);
        zahlenListe.add(h);
    }

    public int getZahl(int index) {
        Integer h = zahlenListe.get(index);
        return h.intValue();
    }
}
```

Angabe von `int` nicht möglich!

Konvertierung `int -> Integer`

Konvertierung `Integer -> int`

Autoboxing

- Das Einpacken eines Elementaren Typs in seine Wrapper-Klasse bezeichnet man als **Boxing**, analog wird das Auspacken als **Unboxing** bezeichnet.
- Beides kann auch implizit erfolgen (**Autoboxing**).

Beispiel mit Autoboxing

```
public class ZahlenListe {
    private List<Integer> zahlenListe;

    public ZahlenListe() {
        zahlenListe = new ArrayList<Integer>();
    }

    public void addZahl(int zahl) {
        zahlenListe.add(zahl);
    }

    public int getZahl(int index) {
        return zahlenListe.get(index);
    }
}
```

Implizites Boxing

Implizites Unboxing

Autoboxing: Hürden

- Vorsicht: Autoboxing ist lediglich *Syntactic Sugar*. Es schafft die Typsysteme nicht ab.
- Wrapper-Klassen sind Referenztypen, daher ist beim Wertevergleich mit *Schachteln* `equals()` erforderlich, `==` vergleicht i.A. nur die Referenzen!

Formatieren von Zeichenfolgen

- Häufig sind Benutzereingaben Zeichenfolgen, die einem bestimmten Format folgen müssen (z.B. Datum, Gleitkommazahl). Derartige Zeichenfolgen müssen zur Weiterverarbeitung in einen bestimmten Typ umgewandelt werden (z.B. `Date`, `double`).
- Umgekehrt müssen diese Typen in Zeichenfolgen zur Darstellung umgewandelt werden können.

Formatieren von Zeichenfolgen

- Für diese Zwecke bietet Java die Klasse `java.text.Format` samt ihrer abgeleiteten Klassen.
- Konkrete Beispiele:
  - `SimpleDateFormat`
  - `DecimalFormat`

SimpleDateFormat

```
SimpleDateFormat dateformat =
                          new SimpleDateFormat("dd.MM.yyyy");

Date date;

try {
   date = dateformat.parse("1.1.2002");
}
catch (ParseException e) {
   date = null;
}

String datestr = dateformat.format(date);
System.out.println(datestr);
```

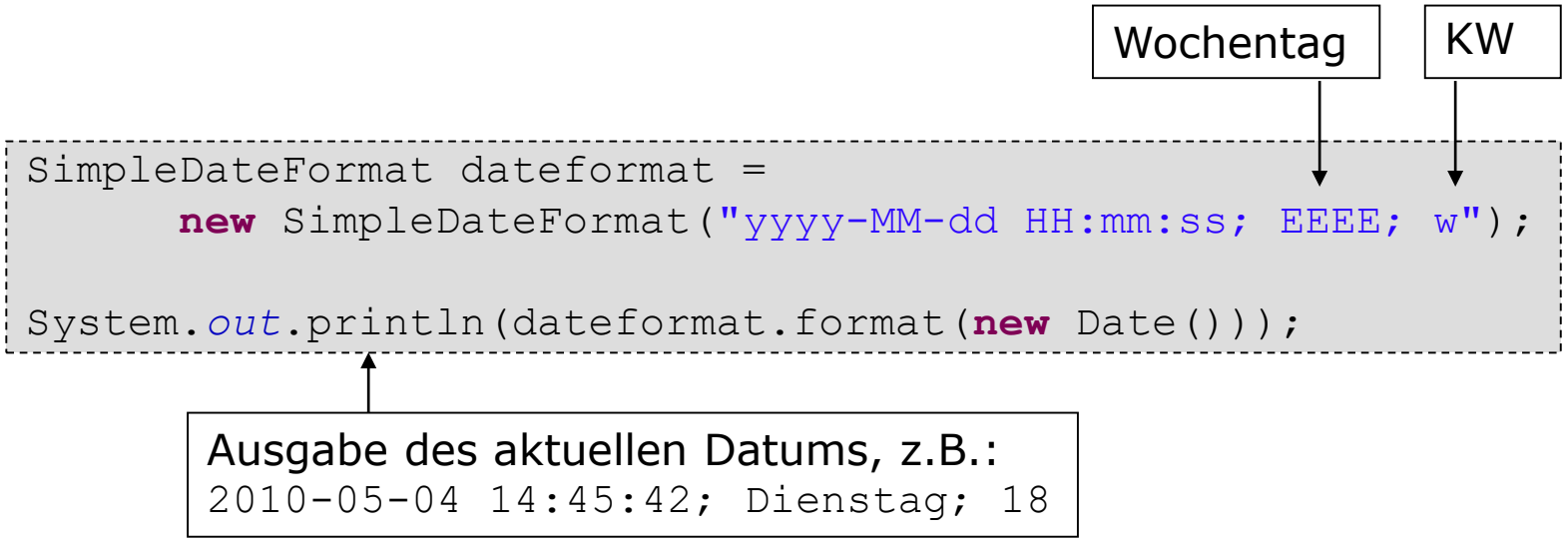
String -> Date

Bei Konvertierungsfehler

Date -> String

Ausgabe: 01.01.2002

SimpleDateFormat



- Die Formatierung wird durch die angegebenen Symbole (z.B. `y`) und deren Wiederholung bestimmt, z.B. liefert `yy` eine zweistellige Jahreszahl und `yyyy` eine vierstellige Jahreszahl.
- Weitere Details siehe: http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html

DecimalFormat

```
DecimalFormat betragformat = new DecimalFormat("#,##0.00");
System.out.println(betragformat.format(2138321366240L));
```

Ausgabe: 2.138.321.366.240,00

- Symboltabelle:

Symbol	Bedeutung
0	Ziffer
#	Ziffer; führende Nullen oder angehängte Nullen im Nachkommabereich werden nicht angezeigt
.	Dezimaltrennzeichen
,	Gruppierungstrennzeichen, i.d.R. zur Festlegung eines Tausenderpunkts

JFormattedTextField

- Erweitert `JTextField` und erlaubt beliebige Formatierung.
- Die eigentlich Formatierung übernimmt eine Unterklasse von `JFormattedTextField.AbstractFormatter` (z.B. `DateFormatter` oder `NumberFormatter`).
- Der Formatter ist durch ein `Format` parametrisiert.

```
class Entity {
   Date datum;
   double betrag;
}

public class FormatDialog extends JDialog {
   class MyOkHandler implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         entity.datum = (Date)txtDatum.getValue();
         entity.betrag = (Double)txtBetrag.getValue();
         dispose();
      }
   }
   …
   private static final SimpleDateFormat DATE_FORMAT =
                                    new SimpleDateFormat("dd.MM.yyyy");
   private static final DecimalFormat BETRAG_FORMAT =
                                    new DecimalFormat("#,###0.00");

   public FormatDialog(Entity entity) { …
      add(txtDatum = new JFormattedTextField(DATE_FORMAT));
      txtDatum.setValue(entity.datum);
      …
      add(txtBetrag = new JFormattedTextField(BETRAG_FORMAT));
      txtBetrag.setValue(entity.betrag);
      …
   }
}
```

JFormattedTextField

- `JFormattedTextField` setzt standardmäßig bei ungültiger Eingabe seinen Wert auf den letzten gültigen Wert zurück.
- Leere Eingaben werden i.d.R. als nicht gültig interpretiert. Um dies zu umgehen, kann beispielsweise ein spezieller Formatter definiert werden.

Beispiel

- Folgender Formatter erlaubt auch Null-Werte:

```
class NullableDateFormatter extends DateFormatter {

   public NullableDateFormatter(DateFormat format) {
      super(format);
   }

   public Object stringToValue(String string) throws ParseException {
      if (string == null || string.length() == 0) {
          return null;
      }
      return super.stringToValue(string);
   }
}
```

- Es kann auch die Methode `String valueToString(Object value)` überschrieben werden, um das Verhalten beim Setzen von Werten zu modifizieren.