Vorlesungsteil

# Verwendung von Bibliotheken

*Bibliotheken rechnen sich nicht, aber sie zahlen sich aus.*
(unbekannt)

# Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Verwendung von Bibliotheken

- Standardbibliotheken
- Klasse Math
- Zufallszahlen, Verteilungen
- Lineare Algebra – JAMA
- Eigene Bibliotheken erstellen
- Java API
- Javadoc

# Pakete einbinden und verwenden

Um Methoden und Klassen einer Bibliothek in einer Klasse bekannt zu machen, können wir diese gleich zu Beginn explizit einbinden. Hier die Syntax, um eine konkrete Klasse Y einzubinden, die in einem dem System bekannten Ordner x liegt:

```
import x.Y;
```

Wenn wir alle Klassen, die in einem Ordner x liegen einbinden möchten, können wir einen * angeben:

```
import x.*;
```

Schauen wir uns dazu nocheinmal die für uns in diesem Kontext relevanten Zeilen des Einlesebeispiels an:

```
 ...
import java.util.Scanner;

public class DateiAuslesen {
   public static void main(String[] args) {
      ...
      Scanner scanner = new Scanner(new File(dateiName));
      ...
   }
}
```

# Pakete einbinden und verwenden II

Bevor der Inhalt der Klasse DateiAuslesen beschrieben wird, machen wir mit dem Schlüsselwort import die Klasse Scanner für unsere Klasse bekannt, die in einem Verzeichnis util liegt, das widerum in einem dem System bekannten Verzeichnis java zu finden ist. Wir haben die Klasse Scanner eingebunden und können diese verwenden.

Alternativ können wir eine Klasse Y aus einem Verzeichnis x direkt ansprechen, ohne das Schlüsselwort import zu verwenden:

```
 ...
public class DateiAuslesen {
   public static void main(String[] args) {
      ...
      java.util.Scanner scanner =
         new java.util.Scanner(new File(dateiName));
      ...
   }
}
```

Wenn wir diese implizite Variante einsetzen, müssen wir allerdings an jeder Stelle den kompletten Pfad angeben.

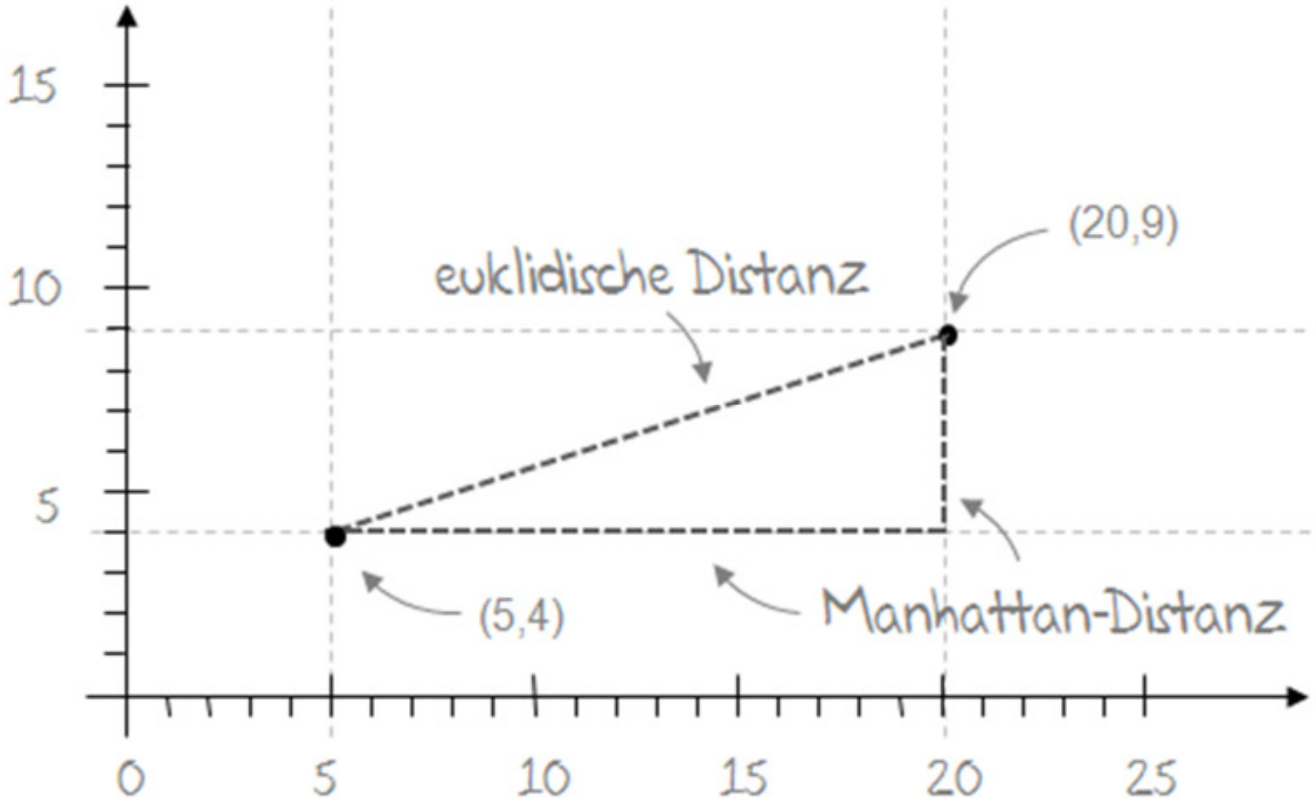
# Methoden der Klasse Math

Die Methoden der Klasse Math werden häufig benötigt. So bietet die Klasse neben einigen trigonometrischen Funktionen, wie sin und cos, auch Funktionen an, wie min, max und log. Es gibt noch sehr viele weitere. Auch die Konstanten PI und E lassen sich dort entdecken.

# Einfache Abstände berechnen

Für die folgenden Beispiele wollen wir uns die Klasse Point aus dem Paket java.awt ausleihen. Ein Exemplar dieser Klasse Point repräsentiert einen zweidimenionalen Punkt bestehend aus einem x- und einem y-Wert.

Wir wollen im folgenden Beispiel den euklidischen Abstand zwischen zwei gegebenen Punkten berechnen:



# Einfache Abstände berechnen II

Da die Methoden in der Klasse Math statisch definiert sind, können alle direkt verwendet werden, wie in dem folgenden Beispiel gezeigt:

```
import java.awt.Point;

public class MathematikTest {
   public static double eukDistanz(Point p1, Point p2) {
      // wir quadrieren die Differenz beider x-Werte
      double diffX = Math.pow(p2.x-p1.x, 2);
      double diffY = Math.pow(p2.y-p1.y, 2); // ... und beider y-Werte

      // schließlich wird die Quadratwurzel aus der Summe
      // der beiden Ergebnisse gezogen
      double dist = Math.sqrt(diffX + diffY);
      return dist; // ... und zurück geliefert
   }

   public static void main(String[] args) {
      Point p1 = new Point( 5, 4);
      Point p2 = new Point(20, 9);

      System.out.println("Euklidische Distanz: "+eukDistanz(p1, p2));
   }
}
```

# Euklidische Distanz und Manhattan-Distanz

Dieses Beispiel berechnet über die Methode eukDistanz den euklidischer Abstand zwischen zwei Punkten p1 und p2:

$$eukDistanz(p_1, p_2) = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2}$$

Als zweite Abstandsmetrik wollen wir kurz den Manhattan-Abstand vorstellen, der die Distanz zwischen zwei Punkten über die Summe der absoluten Differenzen der Einzelkoordinaten angibt:

$$manDistanz(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y|$$

Für die Umsetzung können wir die Methode abs einsetzen:

```
public static double manDistanz(Point p1, Point p2) {
   return Math.abs(p2.x-p1.x) + Math.abs(p2.y-p1.y);
}
```

Die Bezeichnung Manhattan-Abstand ist auf die Gebäude- und Straßenanordnung im gleichnamigen Stadtbezirk von New York zurückzuführen.

# Fläche und Umfang eines Kreises berechnen

Aus dem Mathematikunterricht wissen wir, dass sich die Fläche A eines Kreises mit dem Radius r wie folgt berechnet:

$$A_{Kreis} = \pi r^2$$

Wir können dafür die in der Klasse Math definierten Konstante PI und die Methode pow verwenden:

```
public static double flaecheKreis(double r) {
   return Math.PI * Math.pow(r, 2);
}
```

Der Umfang U eines Kreises mit dem Radius r ergibt sich aus:

$$U_{Kreis} = 2\pi r$$

Auch diese Umsetzung ist jetzt sehr einfach:

```
public static double umfangKreis(double r) {
   return 2 * Math.PI * r;
}
```

# Zufallszahlen mit Math

In der Mathebibliothek gibt es mit random auch eine Methode zur Erzeugung von Zufallszahlen, die haben wir bereits in verschiedenen Projekten kennengelernt.

Wie wollen beispielsweise eine gleichverteilte, ganzzahlige Zufallszahl aus dem Intervall [0,9] erzeugen:

```
int zuffi = (int)(Math.random()*10);
```

Eine Verteilung wird als gleichverteilt bezeichnet, wenn jeder Wert der Verteilung die gleiche Wahscheinlichkeit besitzt, gezogen zu werden. Oder allgemeiner eine gleichverteilte, ganzzahlige Zufallszahl aus dem Intervall [n, m] erzeugen, mit n<m:

```
int zuffi = (int)(Math.random()*(m-n)+n);
```

Für den Fall, dass eine reelle Zufallszahl aus dem Intervall [n, m] mit n<m erzeugt werden soll, können wir die Typumwandlung einfach weg lassen:

```
double zuffd = Math.random()*(m-n)+n;
```

Die die Arbeit mit Zufallszahlen aber einen großen Anwendungsbereich hat, bietet die Klasse Random noch mehr Funktionalität an.

# Zufallszahlen mit Random

Es gibt verschiedene Möglichkeiten, Zufallszahlen in Java zu erzeugen. Mit Math.random steht uns bereits eine Methode zur Verfügung. Jetzt wollen wir die im vorhergehenden Abschnitt vorgestellten Beispiele mit der Methoden der Klasse Random umsetzen. Dazu müssen wir zunächst ein Exemplar der Klasse erzeugen:

```
Random gen = new Random();

// gleichverteilt, ganzzahlig aus [0, 9]
int zuffi    = gen.nextInt(10);

// gleichverteilt, ganzzahlig aus [n, m] mit n<m
zuffi        = gen.nextInt(m-n+1)+n;

// gleichverteilt, reell aus [n, m] mit n<m
double zuffd = gen.nextDouble()*(m-n+1)+n;
```

Mit der Methode nextInt(int n) wird eine ganzzahlige Zufallszahl aus dem Bereich [0,n), also 0 inklusive und n exklusive, erzeugt.

Die Methoden nextFloat und nextDouble liefern Zufallszahlen aus dem Bereich [0,n) mit der entsprechenden Bitanzahl. Es gibt noch weitere Methoden in der Klasse Random.

# Simulation: Lotto 6 aus 49

Als kleine Anwendung werden wir eine Lotterieziehung simulieren. Dazu werden wir 6 Zahlen aus dem Bereich [1,49] ohne zurücklegen zufällig ziehen und die Liste der gezogenen Zahlen zurückliefern:

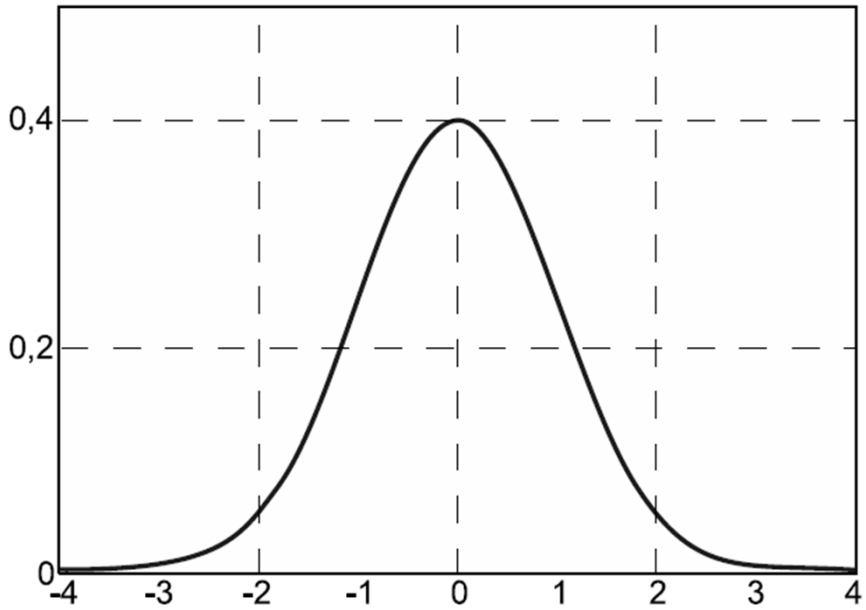
```
public static int[] lotterie() {
   Random rg = new Random();
   int[] zuf = new int[6];
   int wert, i=0;

   aussen:
   while (i<6) {
      wert = rg.nextInt(49)+1;

      // wert in zuf[] schon vorhanden?
      for (int j=0; j<i; j++)
         if (zuf[j] == wert)
            continue aussen;

      zuf[i] = wert;
      i++;
   }
   return zuf;
}
```

So oder so ähnlich könnten die Lottozahlen der nächsten Ziehung aussehen:

```
Lotto (6 aus 49): 16   13   42   40   44   45
```

# Normalverteilte Zufallszahlen

Bisher hatten wir für die gegebenen Zahlenmengen gefordert, dass jede Zahl die gleiche Wahrscheinlichkeit besitzt, gezogen zu werden. Eine weitere sehr wichtige Verteilung ist die Normalverteilung oder Gauß-Verteilung, bei der sich die gezogenen Zufallszahlen in der Mitte der Verteilung konzentrieren und mit größerem Abstand zur Mitte immer seltener auftreten:




Funktionskurve der Standardnormalverteilung mit dem Mittelwert 0 und der Standardabweichung 1. Knapp 70% der Zufallszahlen liegen dann zwischen -1 und 1, 95% zwischen -2 und 2 und fast 100% zwischen -3 und 3.

# Normalverteilte Zufallszahlen I

Die Dichtefunktion der Standardnormalverteilung mit dem Mittelwert 0 und einer Standardabweichung von 1 wird durch folgende Formel beschrieben:

$$f(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}x^2}$$

Wir können beispielsweise mit einem kleinen Würfel-Experiment eine Verteilung erzeugen, die der Normalverteilung sehr ähnlich ist. Gegeben seien dabei zwei Würfel, deren Zahlen gleichverteilt erscheinen. Wir würfeln beide und addieren die Werte. Möglich sind die elf unterschiedlichen Ergebnisse 2 bis 12.

Hier ein kleiner Programmabschnitt, der die Ergebnisse der Einzelexperimente in einer Liste verteilung speichert und zurückgibt:

```
public static int[] wuerfelExperiment(int n) {
   Random rg = new Random();
   int[] verteilung = new int[11];

   for (int i=0; i<n; i++)
      verteilung[rg.nextInt(6) + rg.nextInt(6)]++;

   return verteilung;
}
```

# Normalverteilte Zufallszahlen II

Wenn wir diese Methode nur ein paar mal anwenden (Zeile 1: 10), wirkt die Verteilung zufällig. Bei mehreren Durchläufen (Zeile 2: 100, Zeile 3: 1000) ähnelt diese aber der Normalverteilung mit einem Erwartungswert (so wird der Mittelwert dieser Verteilung genannt) von 7:

```
10:    0    0    3    1    2    2    0    1    0    0    1
100:   1    8    7   10   10   12   18   14    7   10    3
1000: 20   62   93  112  125  171  150  118   70   49   30
```

Für unser Experiment genügte es, Zufallszahlen zwischen 0 und 5 zu erzeugen und diese in eine elf-elementige Liste zu speichern.

# Deterministische Zufallszahlen

Die Erzeugung von Zufallszahlen in einem Computer basiert auf Methoden, die sogenannte Pseudozufallszahlen erzeugen. Dabei ist der Startpunkt einer Zufallszahlenfolge entscheidend. Die Klasse Random bietet beispielsweise zwei Konstruktoren an. Die parameterlose Variante haben wir bereits gesehen. Bei dieser initialisiert sich die Folge in Abhängigkeit zu der aktuellen Systemzeit und erzeugt bei jedem Start neue Folgen von Zufallszahlen.

Um beispielsweise komplexe Programme zu testen, in denen Zufallszahlen eine Rolle spielen, können zufällige Ergebnisfolgen das Debuggen sehr erschweren. Nachdem ein potentieller Fehler identifiziert und korrigiert wurde, sollte eigentlich die gleiche zufällige Ergebnisfolge getestet werden, um sicher gehen zu können, dass der Fehler kein zweites Mal auftritt.

Für solche Fälle gibt es einen parametrisierten Konstruktor mit einem long, bei dem die Folge der Zufallszahlen nach dem Start immer gleich (also deterministisch) ist. Wir können beispielsweise einen long mit dem Wert 0 immer als Startwert, dem so genannten seed-Wert, nehmen und erhalten anschließend immer dieselben Zufallszahlen:

```
long initwert = 0;
Random rGen   = new Random(initwert);
```

Das bedeutet sogar, dass wir den Computer zwischendurch ausschalten können.

# Zufallszahlen sicher und schnell

Ein interessantes Phänomen lässt sich beobachten, wenn wir es darauf anlegen, die größtmögliche Zufallszahl mit der Methode nextDouble aus der Klasse Random zu erzeugen. Wir haben bereits mehrfach erläutert, dass der Wertebereich bei [0,1) liegt, also die 1 nicht erreicht werden kann. Das ist auch soweit richtig, allerdings kann in einem Kontext dieses Ergebnis sehr wohl auftreten.

Angenommen wir haben einen Wert sehr nahe bei 1, z. B. den Wert d in dem folgenden Beispiel:

```
double d = 0.9999999999999999;
System.out.println(d+", "+(d+1));
```

Wenn wir diesen darauf überprüfen, ob der Wert eine 1 darstellt, werden wir immer ein false erhalten. Addieren wir allerdings zu diesem Wert eine 1 erhalten wir eine 2:

```
0.9999999999999999, 2.0
```

Die Methode Math.random basiert intern auf nextDouble und wir können Pech haben, dass der ermittelte Wert so Nahe bei der 1 liegt, dass wir ein unerwartetes Ergebnis erhalten. Sollten wir dann den folgenden, bekannten Weg einschlagen, um ein Zufallszahlintervall zu definieren, kann das fehlschlagen:

```
int zuffi = (int)(Math.random()*n);
```

Wir sollten für diese Fälle also immer die Methode nextInt verwenden.

# Zufallszahlen sicher und schnell II

Ein zweiter wichtiger Punkt beim Umgang mit Zufallszahlen ist sicherlich die Geschwindigkeit. Die Methode random in Math basiert, wie bereits gesagt, auf nextDouble in Random. Daher wird der indirekte Aufruf sicherlich noch etwas Zeit kosten. Mit der Klasse ThreadLocalRandom wird seit Java 1.7 eine Randomklasse angeboten, bei der lediglich ein Exemplar der Klasse verwendet wird (Singleton-Muster):

Folgendes Beispiel führt einen kleinen Performanzvergleich zwischen den drei genannten Varianten durch, bei dem jeweils 10.000.000 Zufallszahlen erzeugt werden:

```
long startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) Math.random();
long stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (Math.random)      "+(stopZeit-startZeit)+" ms");

Random r1 = new Random();
startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) r1.nextDouble();
stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (Random)           "+(stopZeit-startZeit)+" ms");

Random r2 = ThreadLocalRandom.current();
startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) r2.nextDouble();
stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (ThreadLocalRandom)"+(stopZeit-startZeit)+" ms");
```

# Zufallszahlen sicher und schnell III

Das ergab die folgende Ausgabe:

```
Dauer (Math.random)       3984 ms
Dauer (Random)            3610 ms
Dauer (ThreadLocalRandom)1875 ms
```

Wir sehen den großen Zeitunterschied. Demzufolge ist der Einsatz der Klasse ThreadLocalRandom für die Erzeugung von Zufallszahlen zu bevorzugen.

# Live-Coding-Session

# Installation der JAMA-Bibliothek

Um JAMA zu installieren, gehen wir zunächst zu dem aufgeführten Link:
*http://math.nist.gov/javanumerics/jama/*

Wir speichern nun das Zip-Archiv vom Source (zip archive, 105Kb) z.B. in den Ordner "c:\Java\". Jetzt ist das Problem, dass die bereits erstellten .class-Dateien nicht zu unserem System passen. Desshalb müssen die sechs ".class" Dateien im Ordner Jama löschen und zusätzlich das Class-File im Ordner util. Jetzt ist es quasi clean.

Nun gehe in den Ordner "c:\Java" und gebe folgendes ein:

```
C:\Java>javac Jama/Matrix.java
  Note: Jama\Matrix.java uses unchecked or unsafe operations.
  Note: Recompile with -Xlint:unchecked for details.J
```

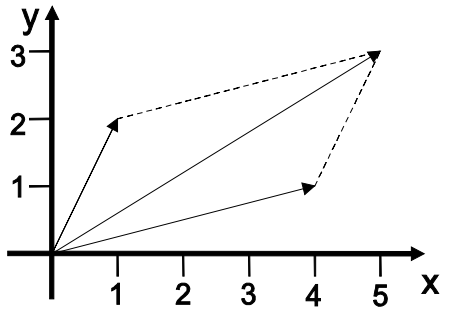
Jetzt wurde das Package erfolgreich compilert.

# Lineare Algebra I

Es gibt viele nützliche Bibliotheken, die wir verwenden können. JAMA ist beispielsweise eine oft verwendete Bibliothek für Methoden der Linearen Algebra.

Hier ein Beispiel zur Vektoraddition:

```
import Jama.*;

public class JAMATest{
   public static void main(String[] args){
      double[][] vector1 = {{1},{2}};
      double[][] vector2 = {{4},{1}};

      Matrix v1 = new Matrix(vector1);
      Matrix v2 = new Matrix(vector2);

      Matrix x  = v1.plus(v2);

      System.out.println("Matrix x: ");
      x.print(1, 2);
   }
}
```



# Lineare Algebra II

Nun wollen wir die Determinante einer Matrix berechnen:

```
import Jama.*;

public class JAMATest{
   public static void main(String[] args){
      double[][] array = {{-2,1},{0,4}};
      Matrix a = new Matrix(array);
      double d = a.det();

      System.out.println("Matrix a: ");
      a.print(1, 2);

      System.out.println("Determinante: " + d);   // det(a) = (-2)*4 - 1*0 = -8
   }
}
```

# Musikdateien abspielen mit AudioClip

Eine einfache Möglichkeit eine Musikdatei abzuspielen, bietet die Klasse AudioClip aus dem Paket applet an. In dem folgenden Beispiel lesen wir die Musikdatei bach.wav ein und spielen diese ab:

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.net.URL;

public class MusikAbspielen {
   public static void main(String[] args) {
      AudioClip sound = null;
      try {
         sound = Applet.newAudioClip(
            new URL("file:///c://bach.wav"));
      }
      catch(Exception ex) {
         ex.printStackTrace();
      }
      sound.play();

      ...
   }
}
```

Dazu haben wir zunächst die Klasse URL aus dem Paket net verwendet, um den Ort einer lokal vorliegenden Musikdatei netzwerkspezifisch anzugeben. Anschließend wurde die Datei mit Hilfe der Klasse Applet aus dem Paket applet geladen. Über die Methode play wird die Datei einmal abgespielt und alternativ in einer Schleife über die Methode loop.

# Eigene Bibliothek erstellen

Die Erzeugung eines eigenen Pakets unter Java ist sehr einfach. Wichtig ist das Zusammenspiel aus Klassennamen und Verzeichnisstruktur. Angenommen wir wollen eine Klasse MeinMax in einem Paket meinMathe anbieten. Dann legen wir ein Verzeichnis meinmathe an und speichern dort z. B. die folgende Klasse:

```
package meinmathe;

public class MeinMax{
   public static int maxi(int a, int b){
      if (a<b) return b;
      return a;
   }
}
```

Durch das Schlüsselwort package signalisieren wir, dass es sich um eine Klasse des Pakets meinmathe handelt. Laut Konvention verwenden wir ausschließlich Kleinbuchstaben. Firmen verwenden beispielsweise oft ihre umgedrehte Internetdomain: com.example.mypackage.

Unsere kleine Matheklasse bietet eine bescheidene maxi-Funktion.

# Eigene Bibliothek erstellen II

Nachdem wir diese Klasse kompiliert haben, können wir außerhalb des Ordners eine neue Klasse schreiben, die dieses Paket testweise verwendet. Dabei ist darauf zu achten, dass der Ordner des neuen Paketes entweder im gleichen Ordner wie die Klasse liegt, die das Paket verwendet, oder dieser Ordner im CLASSPATH aufgelistet ist.

Hier unsere Testklasse:

```
import meinmathe.MeinMax;

public class MatheTester{
   public static void main(String[] args){
      System.out.println("Ergebnis = "+MeinMax.maxi(3,9));
   }
}
```

Wir erhalten nach der Ausführung folgende Ausgabe:

```
C:\JavaCode>java MatheTester
Ergebnis = 9
```

Das kleine Beispiel hat uns gezeigt, wie es mit wenigen Handgriffen möglich ist, unsere Methoden zu Paketen zusammenzufassen. Die Verzeichnisstruktur kann natürlich noch beliebig verschachtelt werden.

# JavaDoc

Eine ausreichende und nachvollziehbare Kommentierung von Programmen ist ein wichtiger und absolut notwendiger Bestandteil bei der Entwicklung von Software. Wir haben mit dem einzeiligen und dem mehrzeiligen Kommentar bereits zwei wichtige Kommentierungsvarianten kennengelernt:

```
// Einzeiliger Kommentar

/* Kommentar geht über mehrere
   Zeilen */
```

Mit JavaDoc gibt es eine weitere Variante, die zusätzlich die Möglichkeit bietet, die kommentierten Klassen und beschriebenen Methoden in ein HTML-Format zu überführen und damit eine Dokumentation der aktuellen Programme vorzunehmen. Wir werden im Folgenden nur eine kurze Einführung geben.

Ein solcher Kommentar beginnt mit /** und endet ebenfalls mit */:

```
/**
 * Ich bin ein besonderer Kommentar
 */
```

Die führenden Sternchen vor jeder Zeile sind in neueren JavaDoc-Versionen nicht mehr notwendig.

# Spezielle Elemente eines Kommentars

```
/**
 * Berechnet den Manhattan-Abstand zwischen zwei Punkten. <p>
 * Die Bezeichnung Manhattan-Abstand ist auf die Gebäude-
 * und Straßenanordnung im gleichnamigen Stadtbezirk von
 * New York zurückzuführen.
 *
 * [@author  ...]
 * [@version ...]
 *
 * @param p1 erster 2d-Punkt
 * @param p2 zweiter 2d-Punkt
 *
 * @return Manhattan-Abstand
 * [@throws ...]
 * [@see    ...]
 */
public static double manDistanz(Point p1, Point p2) {
   ...
```

In der Konsole navigieren wir in den Ordner, dessen Klassen wir dokumentieren wollen. Für unser Beispiel werden wir die kommentierte Klasse MathematikTest verwenden:

```
C:\JavaCode\kapitel8>javadoc MathematikTest.java
```

Vorlesungsteil

# Grafische Benutzeroberflächen

*Mancher kann nicht aus dem Fenster hinausdenken.*
Wilhelm Busch

# Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Grafische Benutzeroberflächen

- Java AWT und Swing
- Fenster erzeugen, positionieren und zentrieren
- Textausgaben und Zeichenfunktionen
- Klasse Color
- Bilder laden und anzeigen
- MediaTracker
- Fensterereignisse
- WindowListener, WindowAdapter
- Innere und anonyme Klassen
- Button, Label und TextField
- Mausereignisse
- Ausblick zu fortgeschrittenen Themen
- Double buffering, Threads, Fullscreen
- Projekt: Analoge Uhr
- Eigene bunte Konsole
- Projekt: Conway's Game of Life
- Projekt: Bildtransformation in Symbolraum

# Ein Fenster erzeugen

Wir können schon mit wenigen Zeilen ein Fenster anzeigen, indem wir eine Instanz der Klasse Frame erzeugen und sie sichtbar machen. Bevor wir die Eigenschaft, „ist sichtbar" mit setVisible(true) setzen, legen wir mit der Funktion setSize(breite, hoehe) die Fenstergröße fest:

```
import java.awt.Frame;

public class MeinErstesFenster {
  public static void main(String[] args) {
     // öffnet ein AWT-Fenster
     Frame f = new Frame("So einfach geht das?");
     f.setSize(300, 200);
     f.setVisible(true);
  }
}
```

Nach dem Start öffnet sich folgendes Fenster:

# Live-Coding-Session

# Fenster erzeugen II

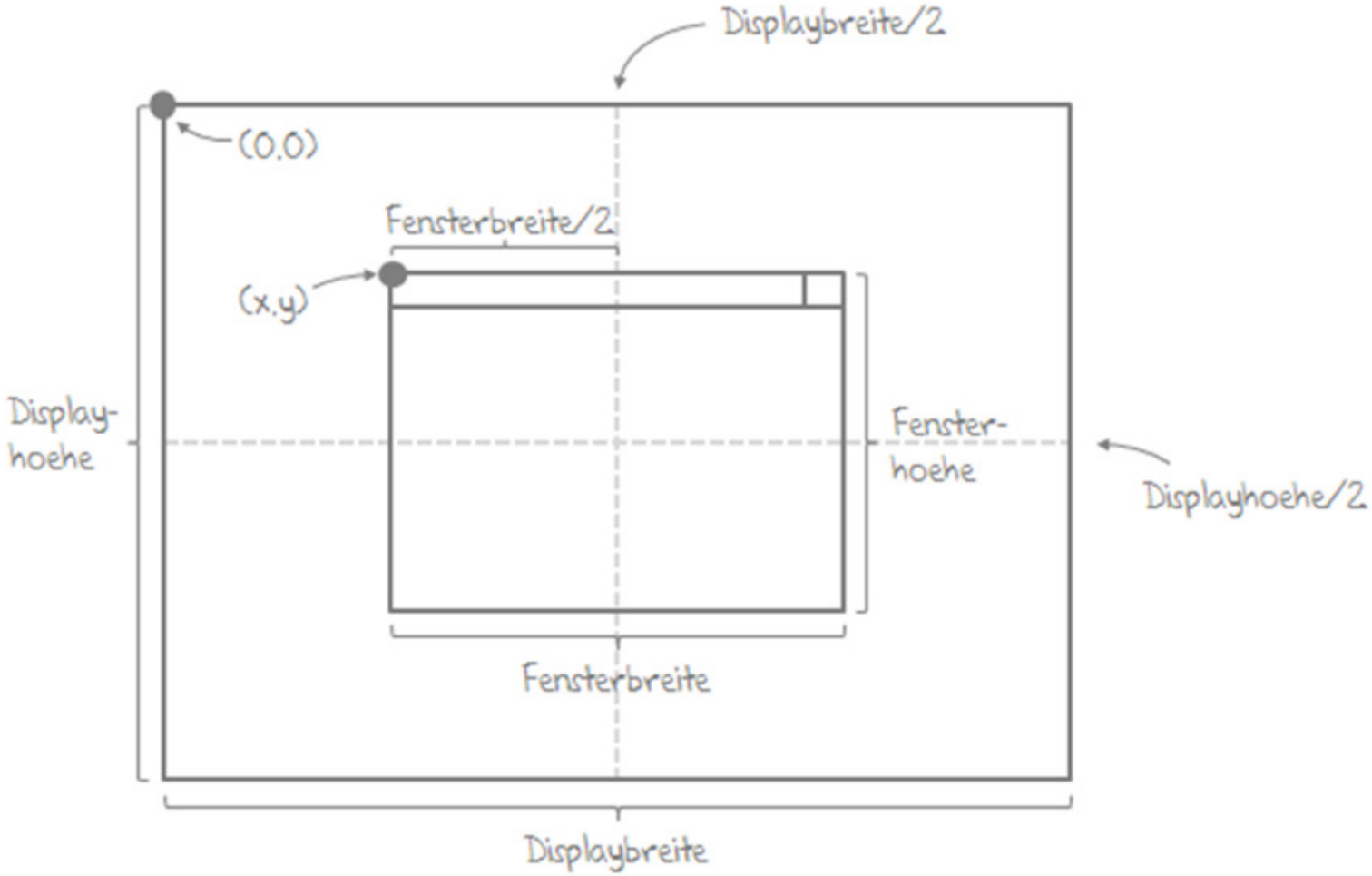
Nach der Erzeugung des Fensters ist die Ausgabeposition die linke obere Ecke des Bildschirms.

Das Fenster lässt sich momentan nicht ohne Weiteres schließen. Wie wir diese Sache in den Griff bekommen und das Programm nicht jedes mal mit Tastenkombination STRG+C (innerhalb der Konsole) beenden müssen, sehen

wir später.

Da das dort vorgestellte Konzept doch etwas mehr Zeit in Anspruch nimmt, experimentieren wir mit den neuen Fenstern noch ein wenig herum.

# Fenster und Screen



# Fenster positionieren

Wir wollen das Fenster zentrieren:

```
import java.awt.*;

public class FensterPositionieren extends Frame {
   public FensterPositionieren(int w, int h){
      setTitle("Ab in die Mitte!");
      setSize(w, h);
      Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
      setLocation((d.width-w)/2,  (d.height-h)/2);
      setVisible(true);
   }

   public static void main( String[] args ) {
      FensterPositionieren f = new FensterPositionieren(200, 100);
   }
}
```

# Fenster positionieren II

Um das Fenster zu zentrieren, lesen wir über die Methode getScreenSize die aktuelle Bildschirmauflösung aus und speichern diese in d. Um ein Fenster zu platzieren, wird die linke, obere Ecke festgelegt. Aktuell steht diese auf (0,0), was die Position erklärt.

Mit der Methode setLocation legen wir den Ankerpunkt des Fensters neu fest und machen es anschließend sichtbar. Alternativ können wir über die Methode setLocationRelativeTo(null) den gleichen Effekt erzeugen.

# Live-Coding-Session

# Das Hollywood-Prinzip

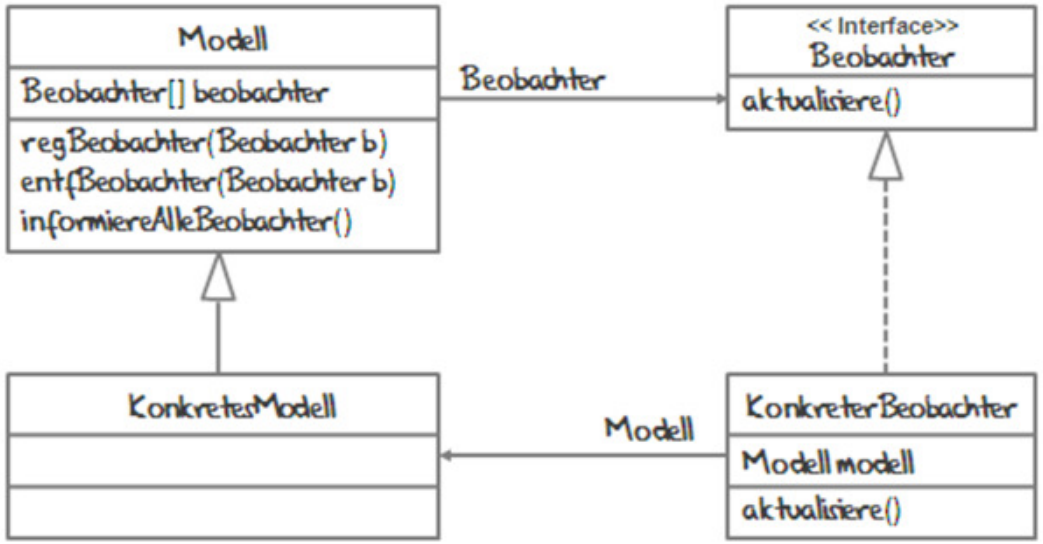
Für die typische Entwicklung von Oberflächen in Java müssen wir uns daran gewöhnen, die Kontrolle abzugeben. Damit ist gemeint, dass Programmteile in den erzeugten Fenstern erst dann ausgeführt werden, wenn eine Aktion seitens des Benutzers ausgeführt wird (z. B. ein Fenster verschieben oder schließen, einen Button drücken, usw.).

Ein derartiges Programmier-Paradigma wird auch als ereignisbasiert bezeichnet (event-driven bzw. event-based programming). Es gilt dabei das Hollywood-Prinzip: „**Don't call us - we'll call you!**".

Wenn mehrere Objekte an dem Zustand eines Objekts interessiert sind und immer erfahren wollen, wann sich dieser geändert hat, haben wir gutes Beispiel für den Einsatz des **Beobachter-Musters**. Sicherlich könnten alle interessierten Objekte ihrerseits permanent den Zustand abfragen, aber die Nachteile sind offensichtlich. Zum Einen gibt es einen enormen Kommunikationsaufwand, der in den meisten Fällen überflüssig ist, und zum Anderen nimmt die Performanz bei steigenden Oberflächenelementen ab.

# Das Bobachter-Muster

In der Softwaretechnik gibt es mit dem Beobachter-Muster (Observer-Pattern) für dieses typische Problem eine akzeptierte Lösung: Das zu beobachtende Objekt bezeichnen wir als Modell (Observables) und die interessierten Objekte als Beobachter (Observer):




Jeder Beobachter, der ab sofort über die Änderungen informiert werden möchte, registriert sich über regBeobachter beim Modell. Sollte sich der Zustand jetzt ändern, so werden alle registrierten Beobachter über informiereAlleBeobachter informiert. Beobachter können sich mit entfBeobachter jederzeit wieder austragen. Für eine konkretes Modell können wir von der Klasse Modell erben. Ein konkreter Beobachter implementiert das Interface Beobachter und bietet die Methode aktualisiere an, die von dem konkreten Modell ausgeführt wird.

Das direkt von Java angebotene Beobachter-Muster wird durch die Klassen Observable und Observer repräsentiert .

# Beispiel mit zwei Beobachtern

Schauen wir uns dazu nur kurz ein Beispiel an, deren Funktionsweise wir in einem der nächsten Abschnitte genauer erläutern werden:

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

public class ObserverAWTBeispiel extends BasisFenster {
   public ObserverAWTBeispiel() {
      super("", 300, 100);

      // Modell Button sendet bei Klick an Beobachter
      Button button = new Button("Mach mich bemerkbar!");

      // Beobachter 1
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            System.out.println("A: Aha, Button geklickt");
         }
      });

      // Beobachter 2
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            System.out.println("B: Aha, Button geklickt");
         }
      });

      this.add(button);
      setVisible(true);
   }
}
```

# Beispiel mit zwei Beobachtern II

Wir haben ein Fenster mit einem Button erzeugt. Wenn dieser Button gedrückt wird, erhalten sowohl Beobachter 1 und 2 diese Information und geben Ihr Benachrichtigung aus:

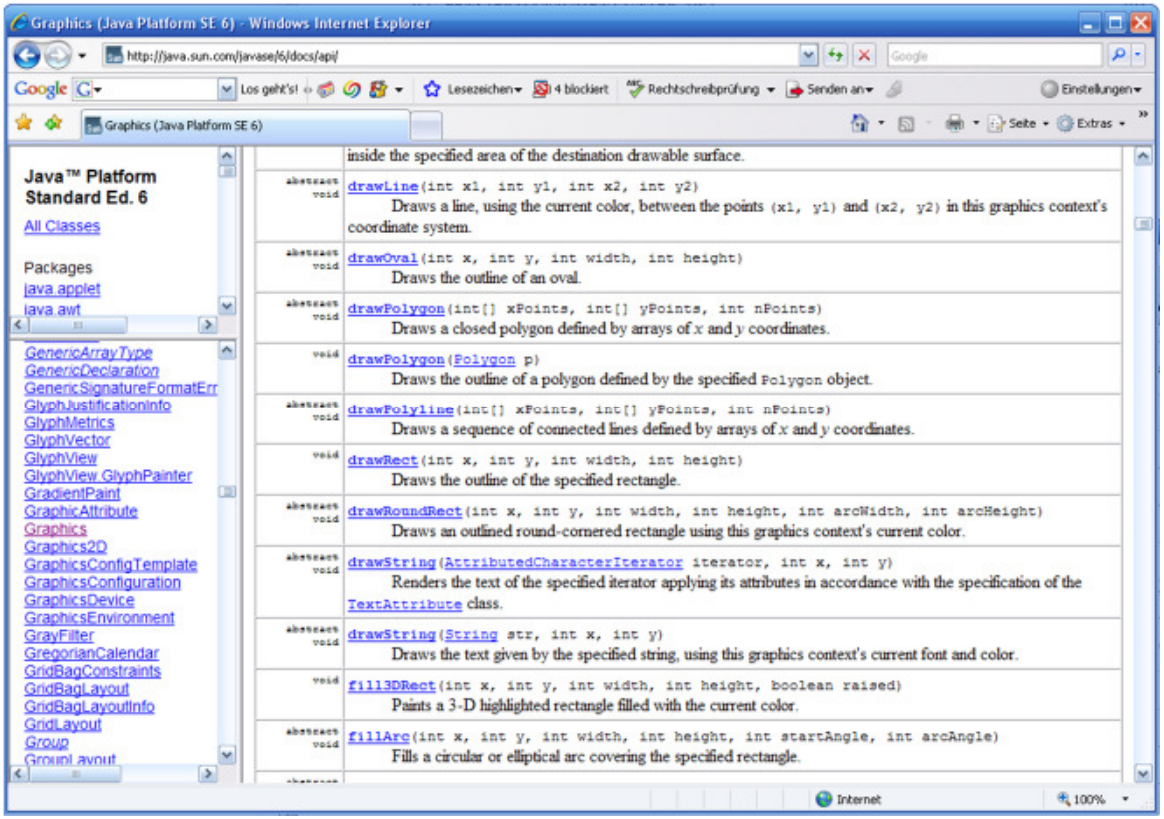
```
A: Aha, Button geklickt
B: Aha, Button geklickt
A: Aha, Button geklickt
B: Aha, Button geklickt
...
```

Es gibt also zwei Klassen, die sich beim Button angemeldet haben erst dann aktiv werden, wenn dieser gedrückt wird. Aber dazu später mehr.

# Methoden der Klasse Frame

AWT bietet eine Reihe von Zeichenfunktionen an. Um etwas in einem Fenster anzeigen zu können, müssen wir die Funktion paint der Klasse Frame überschreiben. Als Parameter sehen wir den Typ Graphics.

Die Klasse Graphics beinhaltet unter anderem alle Zeichenfunktionen. Schauen wir dazu mal in die API:



# Auf Fensterereignisse reagieren

Wir werden jetzt Schritt für Schritt eine nützliche Basisklasse BasisFenster entwickeln, von der wir nur noch ableiten müssen. Das ist unsere Ausgangssituation:

```
import java.awt.Frame;

public class BasisFenster extends Frame {
   public BasisFenster(String titel, int breite, int hoehe) {
      setTitle(titel);
      setSize(breite, hoehe);
      setLocationRelativeTo(null);
      setVisible(true);
   }
}
```

Wir werden das Fenster von einer anderen Testklasse erzeugen und anzeigen lassen:

```
public class FensterTest {
   public static void main(String[] args) {
      BasisFenster fenster = new BasisFenster("Fenster", 300, 100);
   }
}
```

Beim Starten des Programms müssen wir feststellen, dass sich das Fenster nicht über den üblichen Weg schließen läßt. Dazu navigieren wir in die Konsole und beenden das Programm z. B. mit der Tastenkombination STRG+C.

# Das Interface WindowListener

Wir erinnern uns an das besprochene Beobachter-Muster. An dieser Stelle müssen wir einen Beobachter bestimmen, der darüber informiert wird wenn das Fenster geschlossen werden soll und sich entsprechend darum kümmert.

Mit dem WindowListener steht uns ein Interface mit zahlreichen Methoden zur Verfügung, das unter anderem eine Methode windowClosing für das Schließen eines Fensters bereithält. Wir können unsere Klasse um Methoden dieses Interfaces erweitern.

Da es sich um ein Interface handelt, müssen wir alle Methoden selbst implementieren, wie das folgende Beispiel zeigt:

# Das Interface WindowListener II

```
import java.awt.Frame;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class BasisFenster extends Frame implements WindowListener {
   public BasisFenster(String titel, int breite, int hoehe) {
      setTitle(titel);
      setSize(breite, hoehe);
      setLocationRelativeTo(null);

      // WindowListener als Beobachter hinzufügen
      addWindowListener(this);

      setVisible(true);
   }

   // ***********************************************************
   // Hier werden alle WindowListener-Methoden implementiert,
   // obwohl nur windowClosing für uns interessant ist:
   public void windowClosing(WindowEvent event) {
      dispose();      // Fenster schließen
      System.exit(0); // Programm beenden
   }
   public void windowClosed(WindowEvent event) {}
   public void windowDeiconified(WindowEvent event) {}
   public void windowIconified(WindowEvent event) {}
   public void windowActivated(WindowEvent event) {}
   public void windowDeactivated(WindowEvent event) {}
   public void windowOpened(WindowEvent event) {}
   // ***********************************************************
}
```

# Das Interface WindowListener III

Da uns an dieser Stelle nur die Methode windowClosing interessiert hat, wurden die restlichen Methoden mit einem leeren Methodenkörper implementiert.

Sollte beim Programmstart das Ereignis Fensterschließen stattfinden, so werden wir darüber in der Weise informiert, dass die Methode windowClosing ausgeführt wird. Die Methode schließt das Fenster und beendet anschließend das Programm.

# Live-Coding-Session

# Der abstrakte WindowAdapter

Bei der Implementierung eines Interfaces müssen wir immer alle vorhandenen Methoden implementieren, auch wenn wir wie in dem vorhergehenden Abschnitt nur eine Methode nutzen wollen. Eine naheliegende Idee ist es, eine Klasse bereitzustellen, die alle Methoden bereits leer implementiert, um anschließend von dieser abzuleiten und nur die gewünschten Methoden zu überschreiben.

Mehrfachvererbung geht so nicht.

# Der abstrakte WindowAdapter II

Das müssen wir nicht selbst vornehmen, denn mit der bereits vorhandenen abstrakten Klasse WindowAdapter steht uns eine solche zur Verfügung, die unter anderem die Methoden des Interfaces WindowListener bereits leer implementiert. Im Wege steht uns allerdings die Tatsache, dass wir in Java nur von einer Klasse über extends direkt ableiten dürfen:

```
import java.awt.Frame;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class BasisFenster extends Frame, WindowAdapter {
   public BasisFenster(String titel, int breite, int hoehe) {
      ...
   }

   // nur diese Methode aus WindowAdapter überschreiben
   public void windowClosing(WindowEvent event) {
      dispose();      // Fenster schließen
      System.exit(0); // Programm beenden
   }
}
```

# Variante 1: Lokale Klassen

Bisher haben wir eine Klasse mit der Datei in der sie sich befindet gleichgestellt. In Java ist es aber erlaubt, mehrere Klassen in einer Datei unterzubringen (lokale Klassen), wobei eine public sein muss und der Datei damit den Namen gibt.

Da wir die Funktionalität für das Fensterschließen gerne in der zugehörigen Datei halten wollen, können wir eine weitere Klasse in der Datei angeben:

```
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class BasisFenster extends Frame {
   public BasisFenster(String titel, int breite, int hoehe) {
      setTitle(titel);
      setSize(breite, hoehe);
      setLocationRelativeTo(null);
      addWindowListener(new WindowClosingAdapter());
      setVisible(true);
   }
}

class WindowClosingAdapter extends WindowAdapter {
   public void windowClosing(WindowEvent event) {
      event.getWindow().dispose(); // Fenster schließen
      System.exit(0);              // Programm beenden
   }
}
```

Die Klasse WindowClosingAdapter erbt direkt von WindowAdapter und überschreibt die Methode windowClosing. Da die Klasse theoretisch auch mehrere Fenster beobachten kann, ermitteln wir über getWindow in dem erhaltenen WindowEvent das zu schließende Fenster.

# Variante 2: Innere Klassen

Klassen können auch Klassen enthalten, wie sprechen dann von inneren Klassen. Alternativ zur ersten Variante können wir die Klasse WindowClosingAdapter in die Klasse BasisFenster einfügen:

```
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class BasisFenster extends Frame {
   public BasisFenster(String titel, int breite, int hoehe) {
      setTitle(titel);
      setSize(breite, hoehe);
      setLocationRelativeTo(null);
      addWindowListener(new WindowClosingAdapter());
      setVisible(true);
   }

   class WindowClosingAdapter extends WindowAdapter {
      public void windowClosing(WindowEvent event) {
         event.getWindow().dispose(); // Fenster schließen
         System.exit(0);              // Programm beenden
      }
   }
}
```

# Variante 3: Annonyme Klassen

Da wir nur an einer Stelle die Klasse WindowClosingAdapter erzeugen und als Beobachter für das Fensterschließen registrieren, aber später darauf nicht mehr zugreifen, können wir die Klasse auch anonym erzeugen:

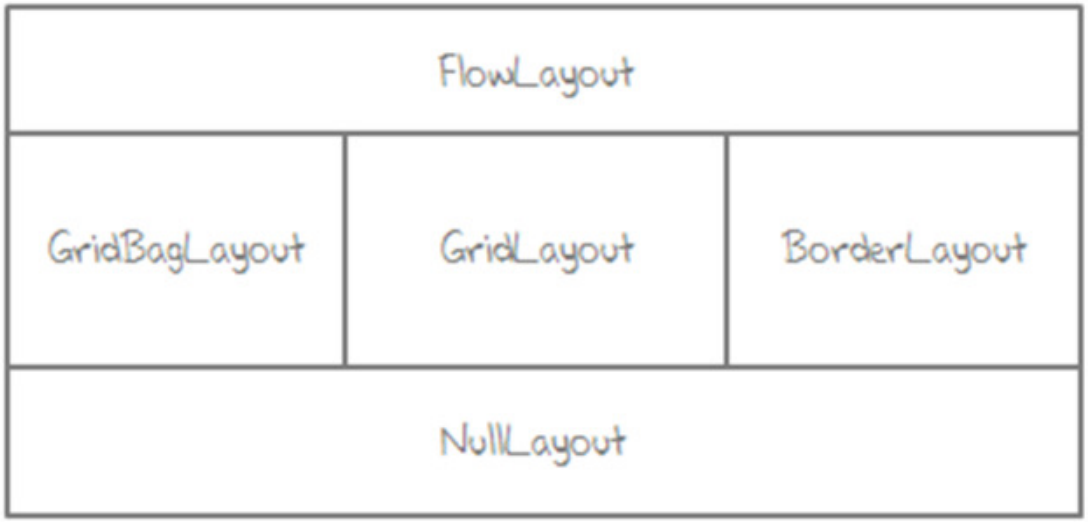
```
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class BasisFenster extends Frame {
   public BasisFenster(String titel, int breite, int hoehe) {
      setTitle(titel);
      setSize(breite, hoehe);
      setLocationRelativeTo(null);
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent event) {
            event.getWindow().dispose(); // Fenster schließen
            System.exit(0);              // Programm beenden
         }
      });
      setVisible(true);
   }
}
```

# Hilfe bei der Layoutgestaltung

Wenn wir typische Elemente wie z. B. Knöpfe oder Texteingaben auf einem Fenster unterbringen wollen, können wir diese durch sogenannte Layoutmanager automatisch platzieren lassen.

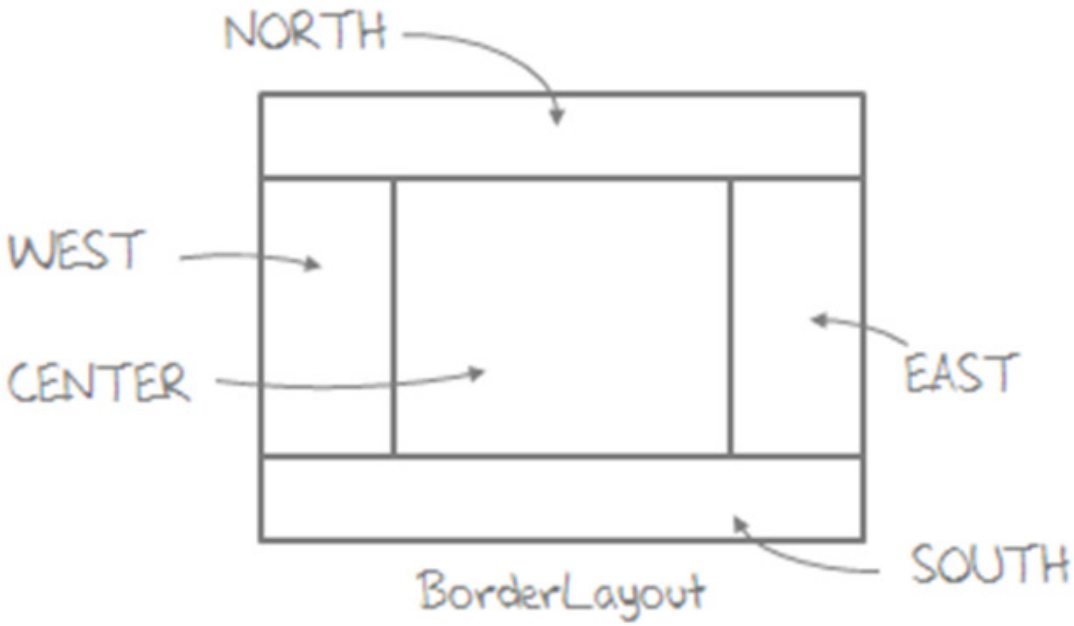
Fenster haben wir bereits kennengelernt. Ein ähnlicher Vertreter ist das Panel, das einen eigenen, viereckigen Bereich definiert. Mehrere Panel können Teil eines Fensters sein. Beide sind sogenannte Container, denen ein Layoutmanager zugeordnet werden kann. Bei AWT gibt es die fünf vordefinierte Varianten Border-, Flow-, Grid-, GridBag- und CardLayout sowie die benutzerdefinierte Variante NullLayout.

Um zu zeigen, dass die Layoutmanager kombiniert werden können, wollen wir das vordefinierte Borderlayout eines Fensters verwenden, um in jedem Bereich eine andere Layoutvariante unterzubringen:



# Borderlayout

Wenn für ein Fenster kein Layoutmanager angegeben wird, verwendet Java standardmäßig das BorderLayout :



# Borderlayout II

Da wir das Schema des BorderLayouts bereits gesehen haben, wollen wir damit anfangen und fünf Label mit den Anfangsbuchstaben der jeweiligen Positionen platzieren:

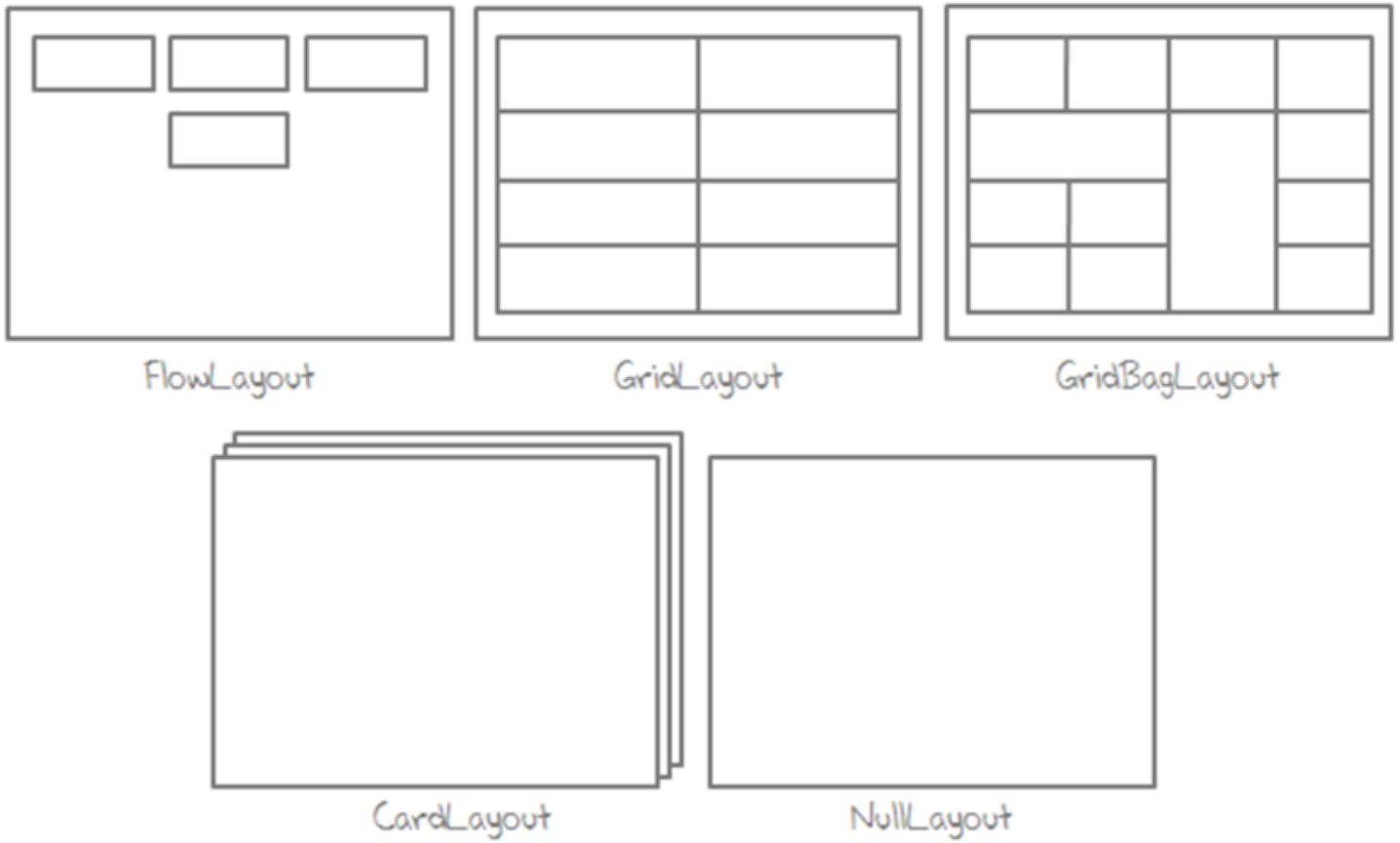
```
private Panel getPanelBorder() {
  Panel panel = new Panel();
  panel.setSize(250, 150);
  panel.setBackground(new Color(150, 150, 150));
  panel.setLayout(new BorderLayout());

  panel.add(new Label("N"), BorderLayout.NORTH);
  panel.add(new Label("W"), BorderLayout.WEST);
  panel.add(new Label("C"), BorderLayout.CENTER);
  panel.add(new Label("E"), BorderLayout.EAST);
  panel.add(new Label("S"), BorderLayout.SOUTH);

  return panel;
}
```

Die Methode getPanelBorder erzeugt zunächst ein Panel der Größe 250x150 (BreitexHöhe) und einem mittleren Grau als Hintergrundfarbe.

Über die Methode setLayout setzen wir einen Layoutmanager für unseren Container. Beim Einfügen eines Fensterelements können wir zusätzlich über die Enumeratoren NORTH, WEST, CENTER, EAST und SOUTH aus der Klasse BorderLayout die Positionierung vornehmen.

# Flowlayout

Bei einem Panel und dem noch später vorgestellten Applet wird standardmäßig der Layoutmanager FlowLayout verwendet:




Die Fensterelemente werden dabei nacheinander in Reihe von links nach rechts angeordnet, wobei eine Reihe immer relativ zur Containerbreite zentriert von oben nach unten angezeigt wird.

# Flowlayout II

Die Methode getPanelFlow erzeugt ein hellgraues Panel der Größe 800x150:

```
private Panel getPanelFlow() {
   Panel panel = new Panel();
   panel.setSize(800, 150);
   panel.setBackground(new Color(200, 200, 200));

   for (int i = 0; i < 10; i++)
        panel.add(new Label("F" + i));

   return panel;
}
```

Es werden anschließend zehn Label mit den Bezeichnern F0, F1, ... , F9 eingefügt.

# Gridlayout

Das GridLayout bietet eine zweispaltige Gitterform an. Beim Hinzufügen der Elemente werden diese zeilenweise aufgefüllt:

```
private Panel getPanelGrid() {
   Panel panel = new Panel();
   panel.setSize(250, 150);
   panel.setBackground(new Color(100, 100, 100));
   panel.setForeground(Color.WHITE);
   panel.setLayout(new GridLayout(4, 3));

   for (int i = 0; i < 12; i++)
        panel.add(new Label("G" + i));

   return panel;
}
```

Ein dunkelgraues Panel der Größe 250x150 mit einem 4x3-Grid wird angelegt. Das bedeutet, es werden vier Zeilen à drei Spalten angelegt. Da zwölf Objekte mit den Bezeichnern G0, G1, ..., G11 eingefügt werden, landen G0, G1 und G2 in der ersten Zeile.

# Gridbaglayout

```
private Panel getPanelGridBag() {
  Panel panel = new Panel();
  panel.setSize(250, 150);
  panel.setBackground(new Color(50, 50, 50));

  GridBagLayout grid = new GridBagLayout();
  panel.setLayout(grid);
  GridBagConstraints gbLayout = new GridBagConstraints();
  gbLayout.fill = GridBagConstraints.BOTH;

  gbLayout.gridx = 0;
  gbLayout.gridy = 0;
  gbLayout.gridheight = 1;
  gbLayout.gridwidth  = 2;
  Label label = new Label("Zwei Spalten");
  label.setBackground(Color.WHITE);
  label.setForeground(Color.BLACK);
  grid.setConstraints(label, gbLayout);
  panel.add(label);

  ... // siehe Code

  return panel;
}
```

# Nulllayout

```
private Panel getPanelNull() {
   Panel panel = new Panel();
   panel.setSize(800, 60);
   panel.setBackground(new Color(250, 250, 250));
   panel.setLayout(null);

   for (int i = 0; i < 15; i++) {
        Label label = new Label("N" + i);
        label.setBounds(i*25, (int)(25 + Math.sin(i)*20), 25, 12);
        panel.add(label);
   }

   return panel;
}
```

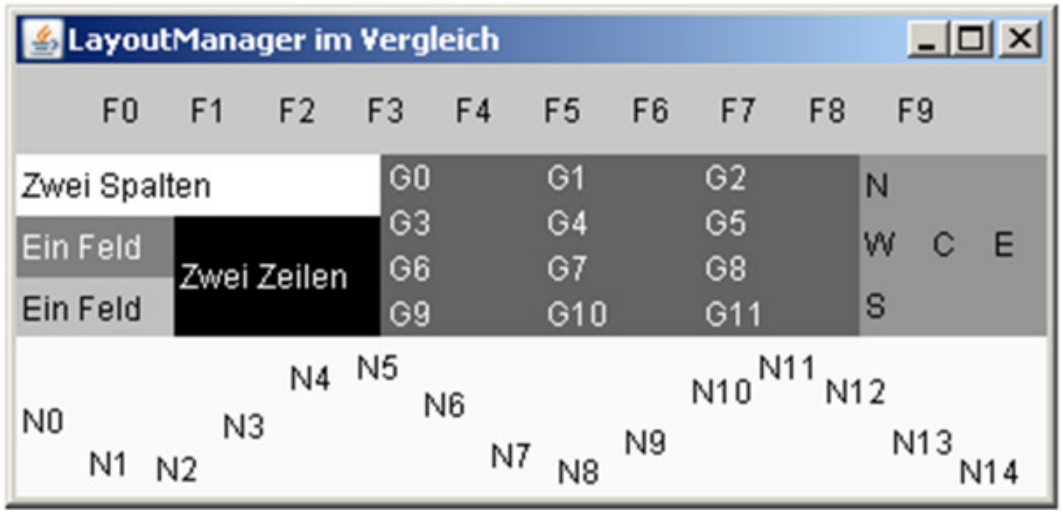
# Alle zusammen

Wir wollen im folgenden Beispiel alle Panels in einem Fenster zusammenfassen, die einzelnen getPanel-Methoden müssen einfach hinzugefügt werden:

```
import java.awt.*;

public class BeispielLayoutmanager extends BasisFenster {
   public BeispielLayoutmanager() {
      super("LayoutManager im Vergleich", 393, 188);

      add(getPanelFlow(),     BorderLayout.NORTH);
      add(getPanelBorder(),   BorderLayout.EAST);
      add(getPanelGrid(),     BorderLayout.CENTER);
      add(getPanelGridBag(),  BorderLayout.WEST);
      add(getPanelNull(),     BorderLayout.SOUTH);

      setVisible(true);
   }

   ... // Layout-Methoden
}
```

# Alle zusammen II

Wir erhalten so eine kleine Übersicht zu den unterschiedlichen Layoutmanagern, die auf diese Weise beliebig verschachtelt und kombiniert werden können:




Alle eingefügten Label sind dabei standardmäßig linksbündig orientiert. Das lässt sich über die Methode setAlignment(int alignment) leicht ändern, wobei Label.LEFT, Label.CENTER und Label.RIGHT vordefinierte Enumeratoren sind.

# Knopf- und Textelemente

Zwei Button und ein Label werden in die GUI eingebettet. Die Anordnung der Elemente innerhalb des Fensters kann von einem Layoutmanager übernommen werden. Für dieses Beispiel haben wir den Layoutmanager FlowLayout verwendet:

```
import java.awt.*;
import java.awt.event.*;

public class BeispielButton extends BasisFenster {
   private Button buttonLinks, buttonRechts;
   private Label  label;

   public BeispielButton(){
      super("Beispiel Knopfverhalten", 500, 70);

      setLayout(new FlowLayout());

      // ActionListener reagiert auf Knopfdruck
      ActionListener aktion = new Knopfdruck();

      // Linker Button erzeugt Kommando "links"
      buttonLinks = new Button("Linker Knopf");
      add(buttonLinks);
      buttonLinks.addActionListener(aktion);
      buttonLinks.setActionCommand("links");

      // Rechter Button erzeugt Kommando "rechts"
      buttonRechts = new Button("Rechter Knopf");
      add(buttonRechts);
      buttonRechts.addActionListener (aktion);
      buttonRechts.setActionCommand("rechts");
```

# Knopf- und Textelemente II

```
      // Label wird das Kommando anzeigen
      label = new Label("Ein Label");
      add(label);

      setVisible(true);
   }

   class Knopfdruck implements ActionListener {
      public void actionPerformed (ActionEvent e){
         label.setText(e.getActionCommand());
      }
   }
}
```

# Live-Coding-Session

# Knopf- und Textelemente III

```
private void buttonLinks() {
   label.setText("Links");
}

private void buttonRechts() {
   label.setText("Rechts");
}

class Knopfdruck implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      // Welcher Button wurde gedrückt?
      String cmd = e.getActionCommand();
      if (cmd.equals("links"))
         buttonLinks();
      if (cmd.equals("rechts"))
         buttonRechts();
   }
}
```

# Live-Coding-Session

# Text eingeben und auslesen

```
import java.awt.*;
import java.awt.event.*;

public class BeispielTextfield extends BasisFenster {
   private Button button;
   private Label label;
   private TextField textfield;

   public BeispielTextfield() {
      super("Beispiel TextField", 500, 70);
      setLayout(new FlowLayout());

      textfield = new TextField("schreib was ...", 15);
      add(textfield);

      ActionListener aktion = new Knopfdruck();
      button = new Button("> Übernimm >");
      add(button);
      button.addActionListener(aktion);
      button.setActionCommand("button");

      label = new Label("... hier landet der Text");
      add(label);
      setVisible(true);
   }

   class Knopfdruck implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         label.setText(textfield.getText());
      }
   }
}
```

# Live-Coding-Session

# Text eingeben und auslesen II

# Text eingeben und auslesen III

```
import java.awt.*;
import java.awt.event.*;
import kapitel3.Codierer;

public class BeispielTextArea extends BasisFenster {
   private TextArea textEingabe, textCodiert, textDecodiert;

   public BeispielTextArea() {
      super("Beispiel TextField", 500, 140);
      setLayout(new FlowLayout());
      textEingabe   = new TextArea("Originaltext", 5, 18, TextArea.SCROLLBARS_NONE);
      add(textEingabe);
      textCodiert   = new TextArea("Codiert", 5, 18, TextArea.SCROLLBARS_NONE);
      textCodiert.setEditable(false);
      add(textCodiert);
      textDecodiert = new TextArea("Decodiert", 5, 18, TextArea.SCROLLBARS_NONE);
      textDecodiert.setEditable(false);
      add(textDecodiert);

      textEingabe.addTextListener(new TextListener() {
         public void textValueChanged(TextEvent e) {
            textCodiert.setText(
               Codierer.codiere(textEingabe.getText(), 21)
            );
            textDecodiert.setText(
               Codierer.codiere(textCodiert.getText(), 21)
            );
         }
      });

      setVisible(true);
   }
}
```

# Live-Coding-Session