Programmieren 2

Fortgeschrittenere Programmierkonzepte

Silko Kruse

Inhalt

- Primitive- & Referenztypen, Felder und Methoden, Zeichenketten, Zweidimensionale Felder
- Komposition, Innere Klassen, Vererbung, Attribute, Polymorphie, Generische Programmierung
- Grafische Bediensysteme, Grundlegende Widgets Ereignisbasierte Programmierung, Ereigniskategorien, Grafische Programmierung
- Container und Komponenten, Ikonen, Ton

Primitive Datentypen und Referenztypen

Primitive Datentypen und Referenztypen

- Datentypen können in zwei Gruppen aufgeteilt werden
  - **Primitive Datentypen**
  - **Referenztypen bzw. abstrakte Datentypen**

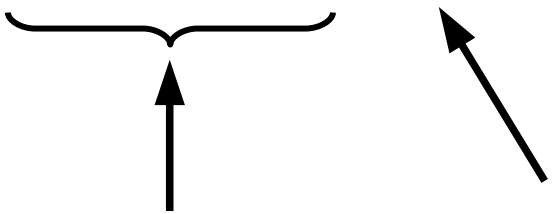
Primitive Datentypen

- **`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`**
  - Werte dieser Typen repräsentieren einen Speicherplatz vordefinierter Größe (Speicherzellen)
  - Speicherplatz wird mit Variablennamen assoziiert
    - Z. B. `int x = 10;`
  - Auf Werte dieser Typen wird direkt zugegriffen
- **`void`**
  - Der Typ **`void`** ("wertlos") bezeichnet eine Methode ohne Rückgabewert

Referenztyp

- Andere Bezeichnung für Klasse
- Auf Werte (Objekte, Instanzen) dieser Typen wird nicht direkt, sondern nur über eine **Referenz** (auch: Verweis, Zeiger) zugegriffen
- Die Referenz entspricht einer Adresse im Hauptspeicher
  - Dies ist die Adresse des Objekts
- Beispiel:

```
LightBulb lb = new LightBulb();
```



Referenztyp Referenzvariable (engl.: *reference variable*)

Referenzvariable

- Die Deklaration einer Variable eines Referenztyps erzeugt KEIN Objekt dieses Typs
- An Stelle dessen wird eine Speicherzelle bereitgestellt, die eine Referenz, d. h. die Speicheradresse eines Objekts dieses Typs, aufnehmen kann
- Beispiel: Fernseher und Fernsteuerung

```
TVObject rc;  // rc referenziert ein
              // Objekt des Typs TVObject
```

Referenzvariablen und Objekte

- `TVObject rc; // Referenzvariable`
- Ein Objekt des Typs muss mit Hilfe der **`new`** Anweisung erzeugt werden
- **`new`** liefert die Referenz, d. h. die Adresse dieses Objekts im Hauptspeicher
- Diese wird in der Referenzvariable gespeichert:

  ```
  rc = new TVObject();
  ```

- Die Referenzvariable kann als eine Art von "Fernsteuerung" interpretiert werden, mit deren Hilfe das erzeugte Objekt benutzt werden kann
- Die Benutzung erfolgt mit Hilfe des Punkt-Operators

Der `null` Wert

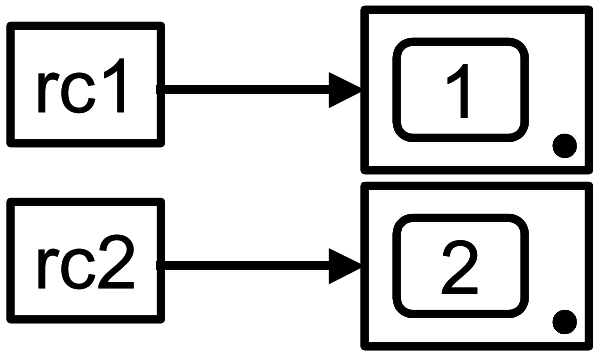
- **`TVObject rc;`**
- Nach der Deklaration der Referenzvariable enthält diese noch keine Referenz auf ein Objekt
- Anstatt dessen enthält diese einen "leeren" Verweis, der durch den speziellen Wert **`null`** repräsentiert wird
- **`null`** bedeutet soviel wie: "keine Referenz auf ein Objekt vorhanden"
- Durch **`rc = new TVObject();`** wird **`null`** durch eine Referenz auf das gerade erzeugte Objekt ersetzt

Mehrere Objekte vom gleichen Typ erzeugen

- Jeder Aufruf von **`new`** erzeugt ein neues Objekt des angeforderten Typs

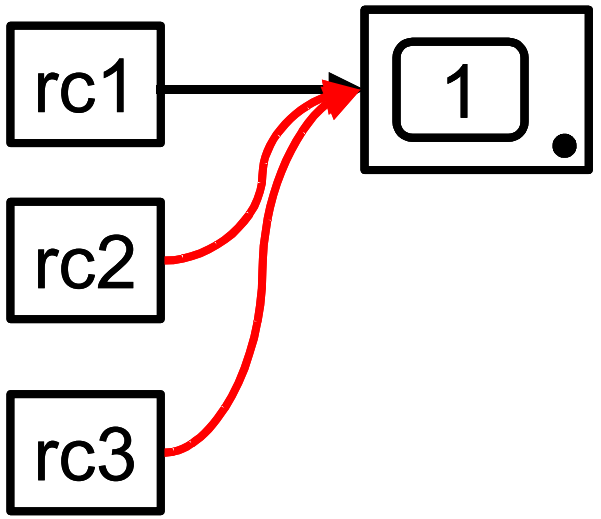
```
TVObject rc1 = new TVObject();
TVObject rc2 = new TVObject();
```

- **`rc1`** und **`rc2`** referenzieren zwei *verschiedene* Objekte vom Typ **`TVObject`**



Mehrere Verweise auf dasselbe Objekt

```
TVObject rc1 = new TVObject();
TVObject rc2 = rc1;
TVObject rc3;
rc3 = rc1;
```



- Das Zuweisen einer Referenz an weitere Referenzvariablen erzeugt zusätzliche Referenzen für dasselbe Objekt
- Alle Referenzvariablen oben verweisen auf dasselbe Objekt

Verlorene Objekte

```
TVObject rc1 = new TVObject();
TVObject rc2 = new TVObject();
rc2 = rc1;
```

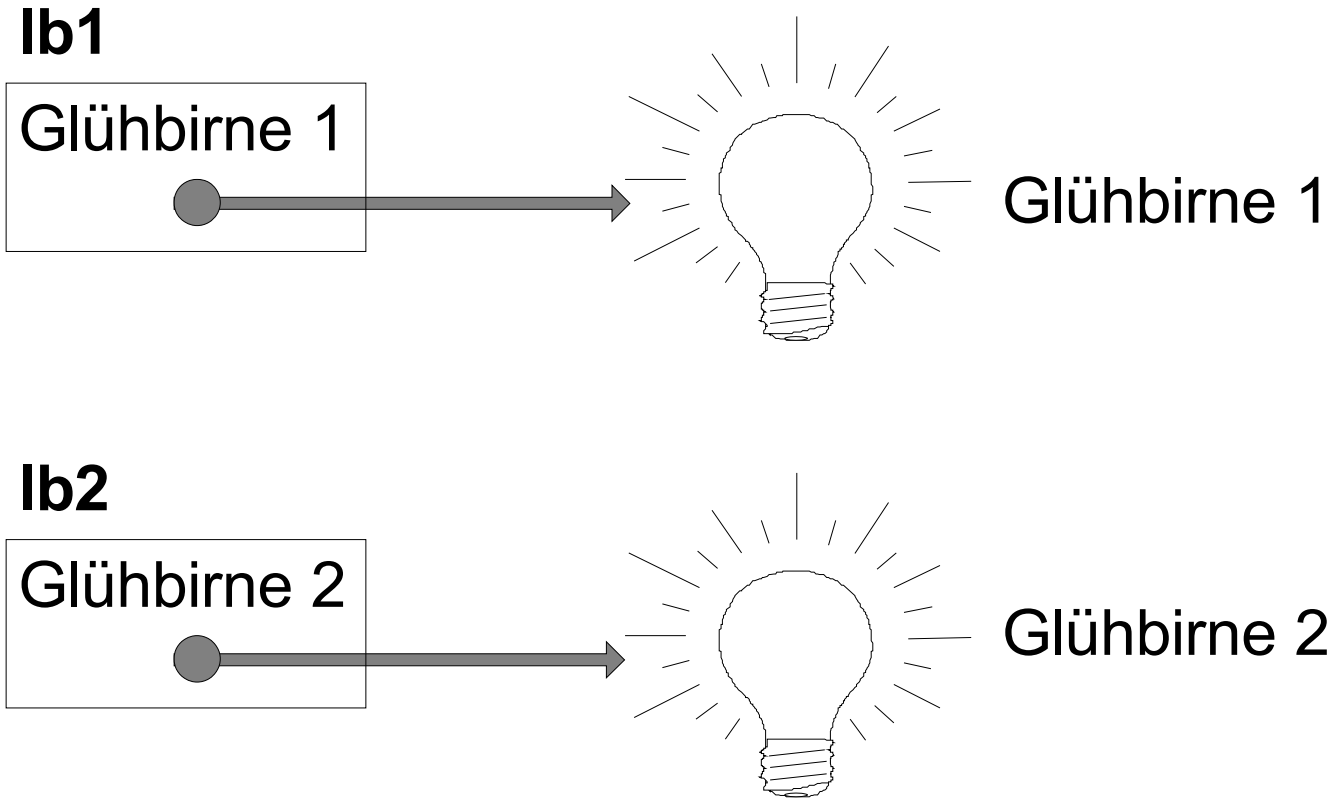
- Das Zuweisen von **`rc1`** an **`rc2`** überschreibt die Referenz auf das zuletzt erzeugte Objekt
- Auf dieses Objekt kann nun nicht mehr zugegriffen werden!
- Dieses Objekt ist nun "Müll" (engl.: *garbage*)

Garbage Collector

- Der Garbage Collector ("Müllsammler") bzw. die **automatische Speicherbereinigung** ist ein Hintergrundprogramm der Laufzeitumgebung
- Der Garbage Collector durchsucht zyklisch den Hauptspeicher nach verloren gegangenen bzw. weggeworfenen Objekten
- Diese werden gelöscht und die von ihnen belegten Speicherzellen dem freien Speicher zugeschlagen

Vergleich von Referenzvariablen (1)

- Vergleichsoperatoren, z. B. "==", vergleichen Referenzen, aber keine Objekte!



Vergleich von Referenzvariablen (2)

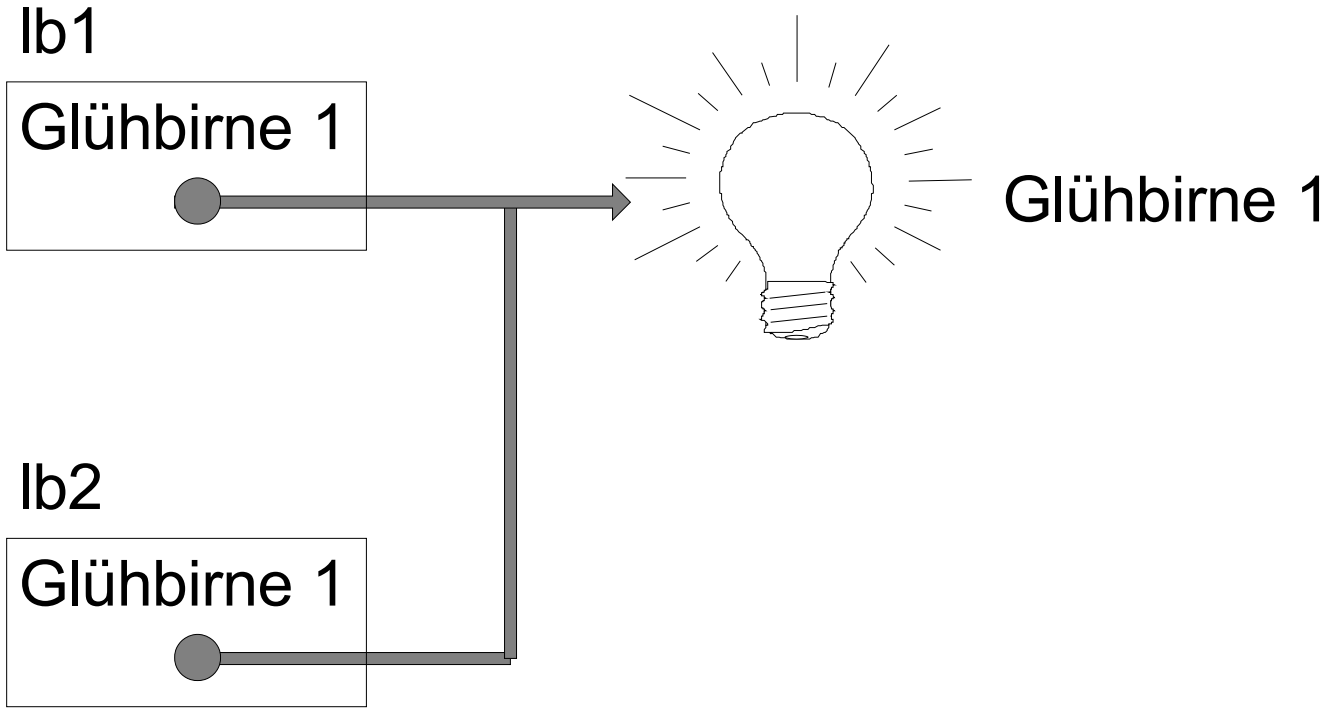
```
// Nachfolgend werden zwei verschiedene Objekte
// mit identischen Eigenschaften erzeugt

LightBulb lb1 = new LightBulb(); // Erzeugt "weiße" Lampe
LightBulb lb2 = new LightBulb(); // Erzeugt "weiße" Lampe

if (lb1 == lb2)
  println("lb1 und lb2 referenzieren das gleiche Objekt");
else
  println("lb1 referenziert ein anderes Objekt als lb2");

// Ausgabe: lb1 referenziert ein anderes Objekt als lb2
```

Vergleich von Referenzvariablen (3)



Vergleich von Referenzvariablen (4)

```
LightBulb lb1 = new LightBulb(); // Erzeugt "weiße" Lampe
LightBulb lb2 = lb1; // Erzeugt zusätzliche Referenz!

if (lb1 == lb2)
  println("lb1 und lb2 referenzieren das gleiche Objekt");
else
  println("lb1 referenziert ein anderes Objekt als lb2");

// Ausgabe: lb1 und lb2 referenzieren das gleiche Objekt
```

Vergleich von Objekten

- Prüfen der Äquivalenz von Objekten, im Hinblick auf deren Eigenschaften, geschieht mit Hilfe der **`equals()`**-Methode
- Standard-**`equals()`**-Methode für alle Klassen bereits vorhanden
  - Diese vergleicht aber **NUR** die Referenzen auf Objekte!
- **`equals()`**-Methode muss deshalb für eigene Klassen durch Überladen selbst definiert werden
- D. h. die für den Vergleich auf Äquivalenz relevanten Anweisungen sind selbst zu implementieren!

Lightbulb mit `equals()`-Methode

```
class LightBulb {

  private boolean light=false;
  private String colour;

  // Code wie zuvor

  LightBulb(String colour) {
    this.colour = colour;
  }

  boolean equals(LightBulb lb) {
    return lb.getColour().equals(colour);
  }

  String getColour() {
    return (colour);
  }

  // Code wie zuvor
}
```

- Hier werden Lampen mit gleicher Farbe als äquivalent definiert
- **`equals`** für Zeichenketten vordefiniert
- Liefert **`true`** falls beide Strings zeichenweise identisch, sonst **`false`**
- Test berücksichtigt Groß- und Kleinschreibung

Vergleich mit `equals()`-Methode

```
LightBulb lb1 = new LightBulb(); // Erzeugt "weiße" Lampe
LightBulb lb2 = new LightBulb("grün"); //Erzeugt "grüne" Lampe

if (lb1.equals(lb2))
  println("Lampen äquivalent");
else
  println("Lampen nicht äquivalent");



// Ausgabe: Lampen nicht äquivalent
```

Referenztypen als Methodenparameter

- Wird ein Objekt als Parameterwert an eine Methode übergeben, so wird nicht das tatsächliche Objekt, sondern nur die Referenz auf das Objekt übergeben
  - Vermeidung unnötigen Kopierens ggf. großer Datenmengen
- Beispiel:

```
MP4Video myVideo = new MP4Video("Dateiname");

resizeMP4Video(myVideo, newSize);
```

- Hier wird nur die Referenz auf das Video übergeben!

Felder und Methoden

Felder als Methodenparameter

- Genauso wie einfache Variablen können auch Felder als Methodenparameter benutzt werden

```
void meineMethode(typ_1[] P_1,...,typ_n[] P_n) {
  // Anweisungen
}
```

Parametertyp Parametername

- Parameter können auch gemischt werden, d. h. nicht alle Parameter müssen Felder oder einfache Variablen repräsentieren

```
void meineMethode(typ_1[] P_1,...,typ_n P_n) {
  // Anweisungen
}
```

Feldelementwerte lesen

- Beispiel:

```
class ArrayMethods { // in ArrayMethods.java
  static void print(int[] x) {
    for (int i = 0; i < x.length; i++)
      System.out.print(x[i] + " ");
    System.out.println();
  }
  static int sumElements(int[] nums) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++)
      sum += nums[i];
    return sum;
  }
}
```

- In Java-Dateien können die Kurzformen **`print`** und **`println`** nicht verwendet werden
- Wir verwenden **`static`**, um ohne Instanz arbeiten zu können

Seiteneffekte

- Da auf Felder mit Hilfe von Referenzvariablen zugegriffen wird, wird dabei nicht das Objekt selbst, sondern nur ein Verweis auf das Objekt an die Methode übergeben
- Mittels dieses Verweises können die Objekteigenschaften und damit die Feldelemente modifiziert werden
- Die Werte von Feldelementen können also aus einer Methode heraus geändert werden!
- Ein **Seiteneffekt** der Methode

Feldelemente beschreiben

```
// Hauptprogramm in Sketch-Datei:
int[] value = {27, 19, 34, 5, 12};
println("Before:");
ArrayMethods.print(value);       // wie bereits definiert
ArrayMethods.zeroElt(value, 0); // siehe unten
println("After:");
ArrayMethods.print(value);
// Ende des Hauptprogramms

class ArrayMethods { // in ArrayMethods.java
  static void zeroElt(int[] x, int elt) { // neue Methode
    if (-1 < elt && elt < x.length)
      x[elt] = 0;
  }
  // print, sumElements wie zuvor
}
```

Ausgabe:

```
Before:
27 19 34 5 12
After:
0 19 34 5 12
```

Felder als interne Variablen

- Felder können auch als interne Variable einer Methode genutzt werden
- Die Referenzvariable des Feldes verliert dann ihre Gültigkeit, wenn die Methode terminiert
  - Das zugehörende Objekt wird vom Garbage Collector gelöscht
- Beispiel:

```
void methodWithInternalArray (int noElements) {

    int[] num = new int[noElements];

    // weiterer Programmcode

}
```

Felder als Rückgabewert (1)

- Felder können auch Rückgabewert einer Methode sein
- Beispiel:

```
// Hauptprogramm in Sketch-Datei:

int[] value = {27, 19, 34, 5, 12};

// Neue Variante der Methode zeroElt:

int[] newArray = ArrayMethods.zeroElt(value, 0);

println("New Array:");
ArrayMethods.print(newArray);

// Ende des Hauptprogramms


// print wie zuvor definiert
```

Ausgabe:

```
New Array:
0 19 34 5 12
```

Felder als Rückgabewert (2)

```
class ArrayMethods { // in ArrayMethods.java

  static int [] zeroElt(int[] x, int elt) {

    int[] modX = new int[x.length];

    for (int i = 0; i < x.length; i++)
      modX[i] = x[i];

    if (-1 < elt && elt < x.length)
      modX[elt] = 0;

    return modX;
  }

  // print, sumElements wie zuvor
}
```

Zeichenketten

Zeichenketten

- Zeichenketten stellen unter Processing bzw. Java Objekte dar
  - D. h. sie sind Instanzen einer Klasse
- Diese Klasse heißt **`String`**
- Entsprechend werden neue Strings mit **`new`** erzeugt:

```
String str = new String("Example String");
```

Vergleich von Zeichenketten

- Da Zeichenketten Objekte sind, stellen zwei identische Strings zwei verschiedene Objekte dar
  - Die zugehörenden Variablen sind Referenzvariablen!
- Beispiel (Hauptprogramm in Sketch-Datei):

```
String strA = new String("Example String");
String strB = new String("Example String");

if (strA == strB)
  println("String A und B: identisch");
else
  println("String A und B: verschieden");


// Ausgabe: String A und B: verschieden
```

Test auf Äquivalenz

- `"=="`-Operator vergleicht Referenzen auf Zeichenketten, aber nicht die Zeichenketten selbst
- Echter Vergleich möglich durch **`equals`**-Methode
  - Diese ist für Zeichenketten schon in geeigneter Weise vordefiniert
  - Führt zeichenweise einen Test auf Übereinstimmung durch
    - Test berücksichtigt Groß- und Kleinschreibung
  - Liefert **`true`**, falls beide **`String`**-Objekte identische Zeichenketten repräsentieren, sonst **`false`**
  - Vergleich ohne Berücksichtigung von Groß- und Kleinschreibung:

    `equalsIgnoreCase(String anotherString)`

Test auf Äquivalenz: Beispiel

- Beispiel (Hauptprogramm in Sketch-Datei):

```
String strA = new String("Example String");
String strB = new String("Example String");

if (strA.equals(strB))
  System.out.println("String A und B: identisch");
else
  System.out.println("String A und B: verschieden");

// Ausgabe: String A und B: identisch
```

Kopieren von Zeichenketten

- Hier gilt dasselbe wie für Objekte: Kopieren der Variableninhalte kopiert nur die Referenz!
- Beispiel:

```
String strA = new String("Example String");

String strB = strA;

// Dies erzeugt zwei Referenzen auf das gleiche Objekt
```

- Um eine Kopie eines Strings zu erzeugen, muss ein neuer String mit der Zeichenkette des zu kopierenden Strings erzeugt werden

Kopieren von Zeichenketten

- Beispiel (Hauptprogramm in Sketch-Datei):

```
String strC = new String("Example String");
String strD = new String(strC);

if (strC == strD)
  println("C und D: gleiches Objekt");
else
  println("C und D: verschiedene Objekte");

// Ausgabe: C und D: verschiedene Objekte

if (strC.equals(strD))
  println("C und D: gleiche Zeichenketten");
else
  println("C und D: verschiedene Zeichenk.");

// Ausgabe: C und D: gleiche Zeichenketten
```

Zeichenketten-Literale

- Da Zeichenketten sehr häufig benutzt werden, steht eine abkürzende Notation zur Verfügung:

```
String str = "Example String";
```

- Dies produziert ein **String-Literal**
- Hierfür ist KEIN **`new`** erforderlich!
- String-Literale sind vergleichbar mit Zahlenwerten
- Das zum Literal gehörende Objekt wird bei erneuter Zuweisung zu einer Variable weiter benutzt

Beispiele

```
int var1 = 7;
int var2 = 7;
```

- `var1` ist gleich `var2`

```
String str1 = "Example String";
String str2 = "Example String";
```

`str1` ist gleich `str2!`

- Hier wird nur EIN Objekt erzeugt und zwei verschiedenen Variablen zugewiesen

Vergleich von Zeichenketten-Literalen

- Die Referenzvariablen **`strA`** und **`strB`** verweisen auf das gleiche Objekt
- Beispiel (Hauptprogramm in Sketch-Datei):

```
String strA  = "Example String";
String strB  = "Example String";

if (strA == strB)
  println("A und B: gleiches Objekt");
else
  println("A und B: verschiedene Objekte");


// Ausgabe: A und B: gleiches Objekt
```

Zeichenketten: Spezielle Methoden

- Die **`String`**-Klasse stellt viele Methoden für Standardaufgaben zur Verfügung
  - Diese gelten gleichermaßen für Strings und String-Literale
- Beispielsweise lässt sich die Länge eines Strings mit Hilfe der Methode **`length()`** bestimmen
  - **`length()`** liefert die Anzahl von Unicode-Zeichen im String
- Beispiel:

```
String str = "Example String";
println(str.length());
```

- Ausgabe: **`14`**

Groß- und Kleinschreibung ändern

- **`toLowerCase()`**
  - Wandelt alle Buchstaben im String in Kleinbuchstaben um
- **`toUpperCase()`**
  - Wandelt alle Buchstaben im String in Großbuchstaben um
- Beispiel:

```
String str = "Example String";
String newStr = str.toUpperCase();
println(newStr);
```

- Ausgabe: **`EXAMPLE STRING`**

Strings modifizieren (1)

- Strings werden NIE direkt modifiziert. Sie sind unveränderbar
- Es wird immer eine modifizierte Kopie zurückgegeben
- Allgemeiner Ansatz:

```
modStr = origStr.operation(par1,...,parN);
```

Strings modifizieren (2)

- Führende oder abschließende Leerzeichen beseitigen:

  `trim()`-Methode

- Beispiel:

```
String userData = "      745    ";
String fixed = userData.trim();
```

- Der neue String `fixed` enthält nur noch `"745"`

Strings modifizieren (3)

- Ein Zeichen im String ersetzen:

  `replace(char oldChar, char newChar)`

  - Erzeugt einen neuen String, in dem alle Vorkommen von **`oldChar`** durch **`newChar`** ersetzt wurden
  - Groß- und Kleinschreibung wird dabei berücksichtigt

Strings modifizieren (4)

- Beispiel:

```
String strA = "string A";
String strB = strA.replace('A','B');

println(strA);
println(strB);

// Ausgabe:
string A
string B
```

Strings modifizieren (5)

- Teil-String im String ersetzen:

  `replaceAll(String str, String replacement)`

  - Erzeugt einen neuen String, in dem alle Vorkommen von **`str`** durch **`replacement`** ersetzt wurden

- Beispiel:

```
String a = "string: The string is an object";
String b = a.replaceAll("string","Zeichenkette");
println(a);
println(b);
```

- Ausgabe:

```
string: The string is an object
Zeichenkette: The Zeichenkette is an object
```

Strings modifizieren (6)

- Nur das erste Vorkommen des Teil-Strings ersetzen:

  `replaceFirst(String str, String replacement)`

  - Erzeugt einen neuen String, in welchem nur das erste Vorkommen von **`str`** durch **`replacement`** ersetzt wurde

- Beispiel:

```
String a = "string: The string is an object";
String b = a.replaceFirst("string","Zeichenkette");
println(a);
println(b);
```

- Ausgabe:

```
string: The string is an object
Zeichenkette: The string is an object
```

In Strings suchen (1)

- Prüfen, ob eine Zeichenkette mit einer bestimmten Teil-Zeichenkette (dem "Präfix") beginnt:

  `startsWith(String prefix)`

  - Liefert Wahrheitswert

- Beispiel:
- **`String str = "xy@hs-ulm.de";`**

  `println(str + " starts with 'xy': " +`
  `str.startsWith("xy"));`

- **`Ausgabe:`**

  `xy@hs-ulm.de starts with 'xy': true`

In Strings suchen (2)

- Beispiel:
- ```
  String str = "uvwxy@hs-ulm.de";
  println(str + " starts with 'xy': " +
          str.startsWith("xy"));
  ```
- Ausgabe:

```
uvwxy@hs-ulm.de starts with 'xy': false
```
```

In Strings suchen (3)

- Prüfen, ob eine Zeichenkette mit einer bestimmten Teil-Zeichenkette (dem "Suffix") endet:

  `endsWith(String suffix)`

  - Liefert Wahrheitswert

- Beispiel:

```
String str = "xy@hs-ulm.de";

println(str + " ends with 'de': " +
        str.endsWith("de"));
```

- Ausgabe:

```
xy@hs-ulm.de ends with 'de': true
```

In Strings suchen (4)

- Prüfen, ob eine Teilzeichenkette in einer anderen Zeichenkette enthalten ist:

  ```
  contains(String searchStr)
  ```

  - Liefert Wahrheitswert **`true`**, wenn Teilzeichenkette enthalten ist, sonst **`false`**

- **Beispiel:**

  ```
  String str = "xy@hs-ulm.de";

  println(str + " contains 'ulm': " +
          str.contains("ulm"));
  ```

- Ausgabe:

  ```
  xy@hs-ulm.de contains 'ulm': true
  ```

In Strings suchen (5)

- Prüfen, ob eine Teilzeichenkette in einer anderen Zeichenkette enthalten ist mit Positionsangabe:

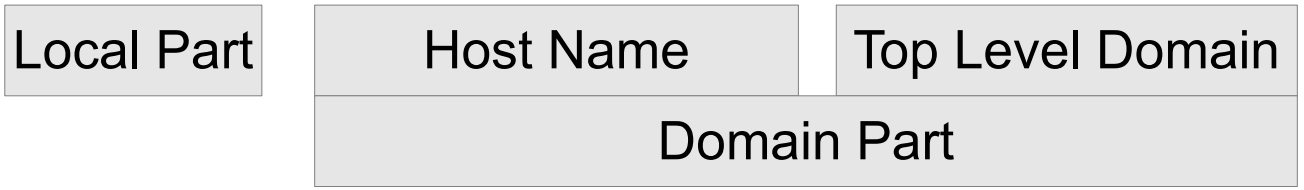
  `indexOf(String searchStr)`

  - Liefert Position des Anfangsbuchstabens der Zeichenkette **`searchStr`** im zugrunde liegenden String
    - Das erste Zeichen im String besitzt die Position 0
  - Ist **`searchStr`** nicht im zugrunde liegenden String enthalten, wird -1 zurückgegeben

In Strings suchen (6)

- Beispiel: Prüfen einer Email-Adresse

```
String str = "brause@wasserundsaft.de";
```




```
println(str + " contains '@': " +
        str.indexOf("@"));

println(str + " contains 'com': " +
        str.indexOf("com"));
```

- Ausgabe:

```
brause@wasserundsaft.de contains '@': 6
brause@wasserundsaft.de contains 'com': -1
```

Teilzeichenkette auslesen

- **`substring(int beginIndex)`**
  - Liefert Teilzeichenkette, die an der Position **`beginIndex`** anfängt und bis zum Ende des Strings reicht
  - Beispiel:

```
String str = "xy@hs-ulm.de";
str.substring(3); // liefert "hs-ulm.de"
```

- **`substring(int beginIndex, int endIndex)`**
  - Liefert Teilzeichenkette, die an der Position **`beginIndex`** anfängt und bis **`endIndex-1`** reicht
  - Beispiel: **`str.substring(0,2); // liefert "xy"`**

Ein einzelnes Zeichen auslesen

- **`charAt(int pos)`**
- Beispiel:

```
String str = "xy@hs-ulm.de";
char ch = str.charAt(2); // liefert ch = '@'
```

String zerlegen: Felder und Strings

- Zeichenkette an der Stelle **`cutter`** aufspalten:

  `split(String cutter)`

  - Erzeugt ein **Feld** von Teil-Strings
- Beispiel:

```
String expression = "12.3 + 3*x + sin(y)";
String[] subexp = expression.split(" [+] ");
println(subexp[0]);
println(subexp[1]);
println(subexp[2]);
```

- Ausgabe:

```
12.3
3*x
sin(y)
```

Zweidimensionale Felder

Zweidimensionale Felder

- Bisher haben wir nur Daten betrachtet, die sich in Form einer Liste darstellen lassen
  - Z. B. Liste zu sortierender Zahlen, jährliche Zinsen, Temperaturverlauf an einem Ort, eine Funktion f(x)
- Viele Daten lassen sich übersichtlicher in Tabellenform darstellen
- Beispiele:
  - Notentabelle
  - Bild
  - Höhenprofil einer Landschaft

Zweidimensionale Felder

- Beispiel: Notentabelle

- Tabellen lassen sich als zweidimensionale Felder darstellen

Person	Klausur			
	0	1	2	3
0	2,7	3,0	3,3	2,3
1	1,3	2,0	1,7	2,3
2	3,0	4,0	3,3	3,7

Zweidimensionale Felder

- Es gelten die gleichen Eigenschaften wie für eindimensionale Felder:
  - Die Feldelemente funktionieren wie normale Variablen, auf die über Indizes zugegriffen werden kann
  - Der Zugriff erfolgt indes über **zwei** Indizes
  - Die Indizes starten immer bei 0 und sind ganzzahlig
  - Alle Feldelemente besitzen den gleichen Datentyp

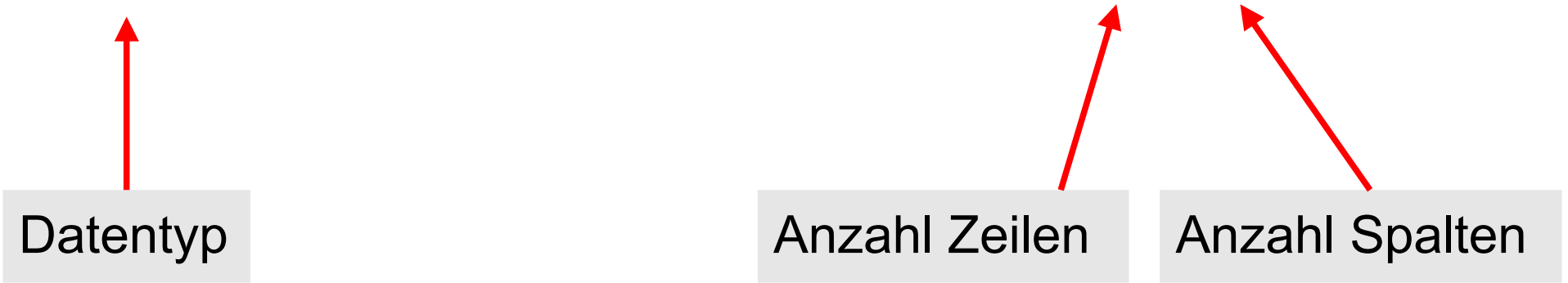
Zweidimensionale Felder

- Beispiel: Notentabelle

Person	Klausur			
	0	1	2	3
0	2,7	3,0	3,3	2,3
1	1,3	2,0	1,7	2,3
2	3,0	4,0	3,3	3,7

- Deklaration:

```
double[][] gradeTable = new double[3][4];
```



Zuweisung

- Beispiel: Notentabelle

```
gradeTable[0][0] = 2.7;
gradeTable[0][3] = 2.3;
gradeTable[1][2] = 1.7;
gradeTable[2][3] = 3.7;
...
```

Person	Klausur			
	0	1	2	3
0	2,7	3,0	3,3	2,3
1	1,3	2,0	1,7	2,3
2	3,0	4,0	3,3	3,7

Ausdrücke

- Es gelten die gleichen Regeln wie für eindimensionale Felder
- Beispiele:

```
// Feldelement in Zeile 3 Spalte 4 inkrementieren
gradeTable[2][3]++ ;

// Feldelement in Zeile 2 Spalte 4 im Zuge der
// Auswertung des Ausdrucks auslesen
double value = (gradeTable[1][3] + 2.0) / 6.0;
```

Ausdrücke

- Beispiele:

```
int i=0, j=1;

// Indexwert(e) aus Variable auslesen
gradeTable[3][j] = 3.3;
double sum = gradeTable[i][j] + gradeTable[i][j+1];

// Indexwert als Wert einer Methode
double value = gradeTable[2][obj.aMethod()] ;
gradeTable[1][0] = gradeTable[i+3][obj.anotherMethod()-2];
```

Deklaration, Erzeugung und Initialisierung

- Auch zweidimensionale Felder bzw. deren Feldelemente können in einem Schritt deklariert, erzeugt und initialisiert werden
- Beispiel:

```
double[][] gradeTable = {
    {2.7, 3.0, 3.3, 2.3},
    {1.3, 2.0, 1.7, 2.3},
    {3.0, 4.0, 3.3, 3.7}
};
```

Zeile 0

Zeile 1

Zeile 2

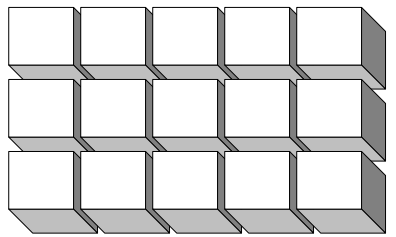
Feld ohne Initialisierungswerte

- Felder stellen Objekte dar

```
int[][] myArray; // Referenzvariable
// Feld-Objekt muss gesondert mit new
// generiert werden!

myArray = new int [3][5];
```

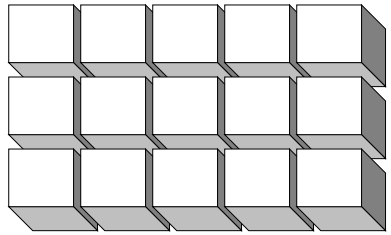
Gleiche Zeilenlängen



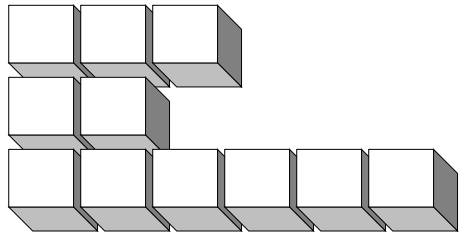
Tabelle

Zeilen unterschiedlicher Länge

- Bisher haben wir nur Felder mit rechteckiger Tabellenstruktur betrachtet



- Die Zeilen eines zweidimensionalen Felds können auch unterschiedliche Längen haben



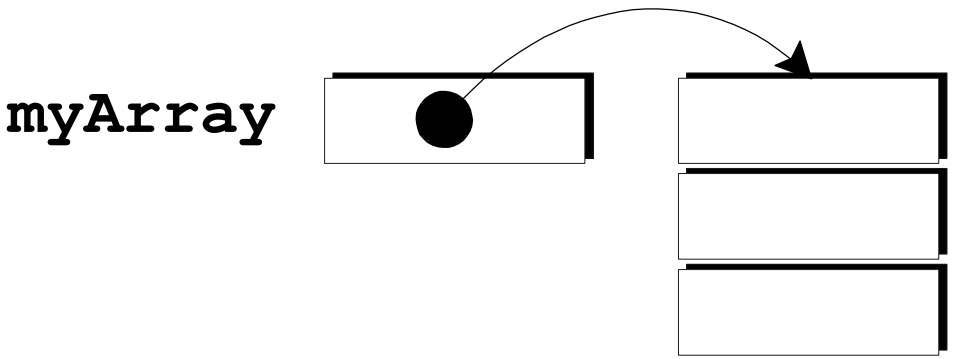
- Wie wird dies programmiert?

Felder als Feldelemente

1. `int[][] myArray;`

Leere Speicherzelle
für Referenz auf
ein zweidimensionales Feld

2. `myArray = new int[3][];`

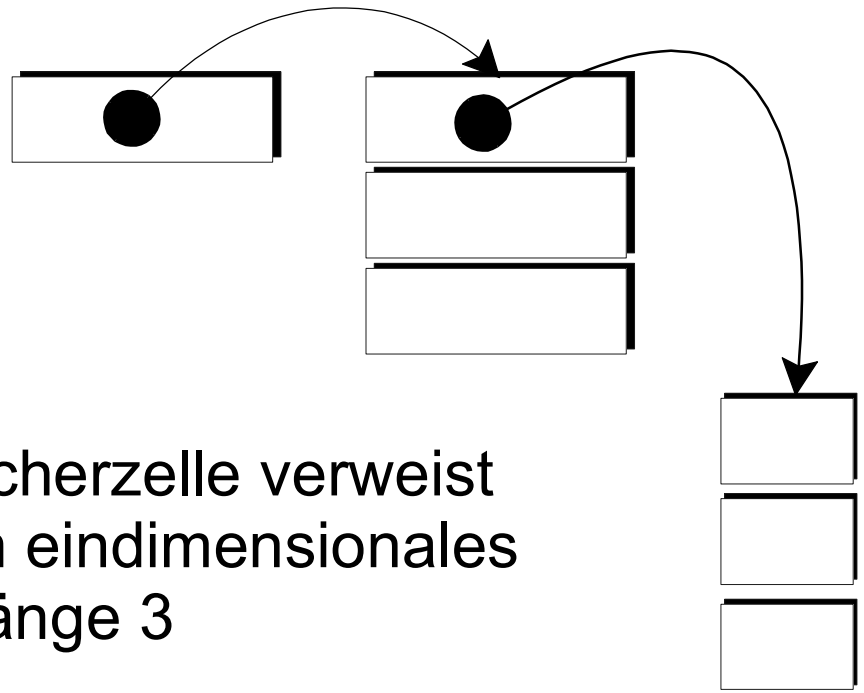


Ein zweidimensionales Feld entspricht einem Feld von Feldern!

1. Speicherzelle enthält jetzt Referenz auf ein Feld
2. Feldelemente dieses Felds referenzieren jeweils ein weiteres eindimensionales Feld

Felder als Feldelemente

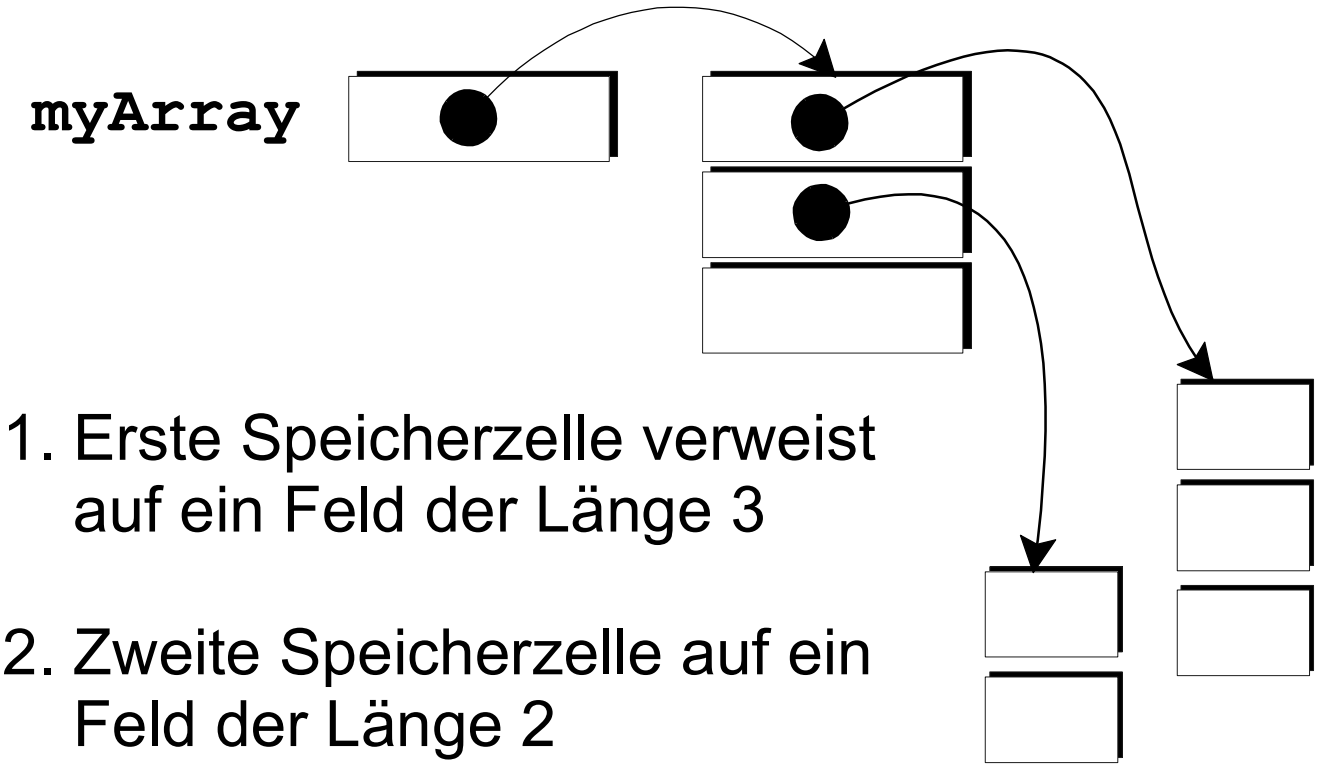
```
3. myArray[0] = new int[3];
```



Erste Speicherzelle verweist nun auf ein eindimensionales Feld der Länge 3

Felder als Feldelemente

```
4. myArray[1] = new int[2];
```

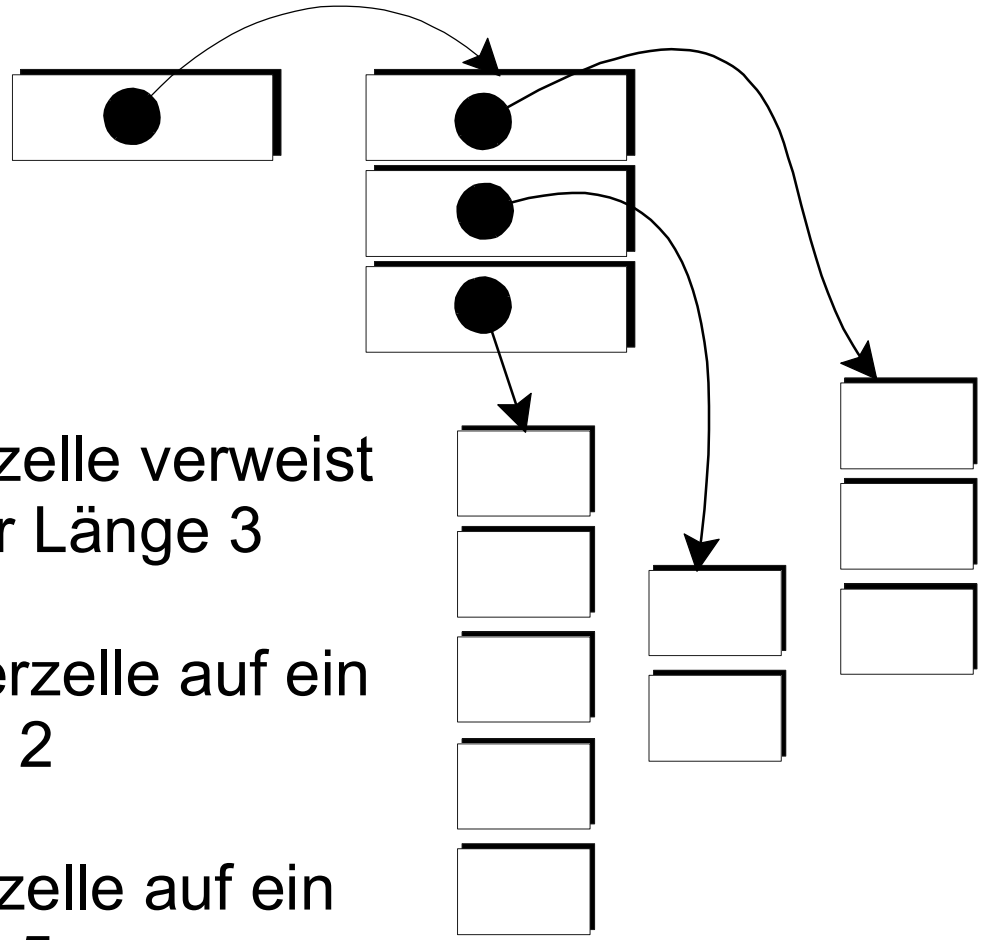


1. Erste Speicherzelle verweist auf ein Feld der Länge 3

2. Zweite Speicherzelle auf ein Feld der Länge 2

Felder als Feldelemente

```
5. myArray[2] = new int[5];
```



1. Erste Speicherzelle verweist auf ein Feld der Länge 3

2. Zweite Speicherzelle auf ein Feld der Länge 2

3. Dritte Speicherzelle auf ein Feld der Länge 5

Zeilen unterschiedlicher Länge: Beispiel

- Sind Initialisierungswerte vorhanden, so kann wieder die abkürzende Notation verwendet werden

```
int[][] arr =
    { { 1, 9, 4 },
      { 0, 2},
      { 0, 1, 2, 3, 4 } };

println("arr[0][2] is " + arr[0][2]); // OK
println("arr[1][1] is " + arr[1][1]); // OK
println("arr[1][2] is " + arr[1][2]); // WRONG!
println("arr[2][4] is " + arr[2][4]); // OK
```

`length`-Wert und Zeilen unterschiedlicher Länge

- Die Länge eines eindimensionalen Feldes ist gleich der Anzahl seiner Feldelemente
- Die Länge eines zweidimensionalen Feldes ist gleich der Anzahl seiner Zeilen
- Dies ist eine Konsequenz aus der möglicherweise unterschiedlichen Anzahl von Spalten pro Zeile

length-Wert: Beispiel

```
int[][] arr =
    {{1, 9, 4},
     {0, 2},
     {0, 1, 2, 3, 4}};
```

Ausgabe:

```
arr[0][2] is 4
arr[1][1] is 2
arr[2][4] is 4
Length is: 3
```

```
println("arr[0][2] is " + arr[0][2]); // OK
println("arr[1][1] is " + arr[1][1]); // OK
println("arr[2][4] is " + arr[2][4]); // OK
println("Length is: " + arr.length);
```

Zeilenlängen

- Da jede Zeile eine andere Länge besitzen kann, steht für jede ein eigener **`length`**-Eintrag zur Verfügung
- Auf diesen wird mit Hilfe des Zeilenindex zugegriffen
- Syntax:

```
Feldname[Zeilennummer].length
```

Zeilenlängen: Beispiel

```
int[][] arr =
    {{1, 9, 4},
     {0, 2},
     {0, 1, 2, 3, 4}};
```

Ausgabe:

```
Length is: 3
Length of row[0] is: 3
Length of row[1] is: 2
Length of row[2] is: 5
```

```
println("Length is: " + arr.length);

// length of each row (number of its columns)
println("Length of row[0]: " + arr[0].length);
println("Length of row[1]: " + arr[1].length);
println("Length of row[2]: " + arr[2].length);
```

Ausgabe der Feldelemente

```
int[][] arr =
     {{1, 9, 4},
      {0, 2},
      {0, 1, 2, 3, 4 }};

for (int row = 0; row < arr.length; row++) {
  print("Row " + row + ": ");
  for (int col = 0; col < arr[row].length; col++)
    print(arr[row][col] + " ");
  println();
}
```

Ausgabe:

```
Row 0: 1 9 4
Row 1: 0 2
Row 2: 0 1 2 3 4
```

Mehrdimensionale Felder

- Wie zweidimensionale Felder mit entsprechend mehr Indizes
- Syntax:

```
Typ[][]...[] Feldname;
Feldname = new Typ[ind1][ind2]...[indN];
```

- Beispiel: 3-dimensionales Feld

```
int[][][] dim3Arr = new int[3][4][7];
```

Komposition und Vererbung

Einführung

- Bis jetzt haben wir Klassen als ein Mittel kennen gelernt Objekte in Software zu modellieren
- Klassen besitzen Variablen, die den Zustand des Objekts codieren
- Diese Variablen heißen Klassenvariablen
- Klassenvariablen können primitive Datentypen besitzen oder Referenztypen
- Wir können also Klassen benutzen, um Klassen zu definieren
- Dies wird als Komposition (engl.: *composition*) bezeichnet

Komposition

Kompositionsbeispiel: Auto bauen

- Erstellt wird eine Klasse für ein Auto
- Das Auto besteht aus Teilen, die für sich selber wieder durch Klassen definiert sind:
- **`Car`**
  - **`Engine`**
  - **`Wheel`**
  - **`Door`**
    - **`Window`**
- **Konvention**: Jede Klasse steht in einer eigenen Datei,

  z. B. **`Engine.java`**
  - Die zum selben Programm gehörenden Dateien bzw. Klassen in das gleiche Verzeichnis legen: Diese werden automatisch von der Java-Laufzeitumgebung gefunden

Motor

```
class Engine {
   public void start() {
      System.out.println("Engine started");
   }
   public void accelerator(int rpm) {
      System.out.println(rpm + "rpm");
   }
   public void stop() {
      System.out.println("Engine stopped");
   }
}
```

Rad

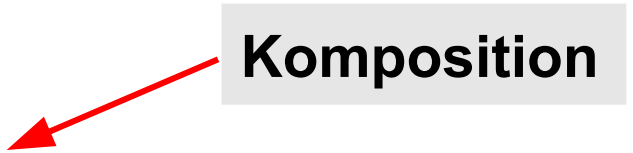
```
class Wheel {
  public void inflate(float pressure) {
    System.out.println("Wheel pressure: " +
                       pressure);
  }
}
```

Fenster

```
class Window {
  public void rollup() {
    System.out.println("Window up");
  }
  public void rolldown() {
    System.out.println("Window down");
  }
}
```

Türen: Komposition

```
class Door {
  public Window window = new Window();
  public void open() {
    System.out.println("Door open");
  }
  public void close() {
    System.out.println("Door closed");
  }
}
```



Auto: Komposition

```
public class Car {
  public Engine engine = new Engine();

  public Door
    left = new Door(),
    right = new Door(); // 2-door

  public Wheel[] wheel = new Wheel[4];

  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
}
```

Hauptprogramm

- Inhalt Sketch-Datei:

```
Car car = new Car();
car.left.window.rollup();
car.wheel[0].inflate(72);
car.left.close();
car.right.close();
```

```
Ausgabe:
Window up
Wheel pressure: 72.0
Door closed
Door closed
```

Klasse Door mit Konstruktor

```
class Door {
  private String whichDoor;
  public Window window = new Window();

  Door (String whichDoor) {
    this.whichDoor = whichDoor;
  }

  public void open() {
    System.out.println(whichDoor + " open");
  }
  public void close() {
    System.out.println(whichDoor + " closed");
  }
}
```

Car mit Konstruktorparameter

```
public class Car {
  public Engine engine = new Engine();

  public Door
    left = new Door("Left door"),
    right = new Door("Right door"); // 2-door

  public Wheel[] wheel = new Wheel[4];

  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
}
```

Neue Ausgabe

- Inhalt Sketch-Datei:

```
Car car = new Car();
car.left.window.rollup();
car.wheel[0].inflate(72);
car.left.close();
car.right.close();
```

```
Ausgabe:
Window up
Wheel pressure: 72.0
Left door closed
Right door closed
```

Innere Klassen

Innere Klassen

- Bisher besaßen Klassen zwei mögliche Komponenten:
  - Variablen und Konstanten
  - Methoden
- Klassen dürfen zusätzlich Klassendefinitionen enthalten
- Derartige Klassen werden als innere Klassen (Engl.: *Inner Classes*) bezeichnet (auch: eingebettete Klassen)

Innere Klassen: Beispiel

```
class EnclosingClass{
   . . .
   class InnerClass {
   . . .
  }
}
```

Warum innere Klassen?

- Zweckmäßig, um die enge inhaltliche Verbindung zweier Klassen zu dokumentieren
- Eine Klasse sollte innerhalb einer anderen Klasse definiert werden, wenn:
  - sie nur sinnvoll im Kontext der umgebenden Klasse angewendet werden kann
  - ihre korrekte Funktion von der umgebenden Klasse abhängt

Benutzung innerer Klassen

- Innere Klassen werden in der üblichen Weise instanziert
- Beispiel:

```
class Parcel {
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    Destination d = new Destination(s);
    return d;
    // return new Destination(s);
  }
}
```

Private innere Klassen

- Ist die Benutzung einer Klasse ausschließlich innerhalb einer anderen Klasse sinnvoll bzw. möglich, so kann man diese durch das Attribut `private` auch völlig einkapseln
- Es ist dann außerhalb der äußeren Klasse nicht mehr möglich auf die Komponenten der inneren Klasse zuzugreifen
- Eine weitere Möglichkeit des *Information Hidings*

Zugriff auf die äußere Klasse

- Die eingebettete Klasse hat vollständigen Zugriff auf alle Komponenten ihrer äußeren Klasse und umgekehrt
- Auch auf die als **`private`** deklarierten Komponenten!
  - Dies ist eine Konsequenz davon, dass die Klasse innerhalb und nicht außerhalb der äußeren Klasse definiert wurde

Processing und innere Klassen

- Die Sketch-Datei eines Hauptprogramms repräsentiert eine Klasse
  - Diese wird nur nicht explizit im Quellcode notiert
- Diese Klasse kann innere Klassen besitzen

Sketch-Datei mit innerer Klasse: Beispiel

```
class LightBulb { // Innere Klasse
  private boolean light = false;
  LightBulb() {
    light = false;
  }
  void switchOn() { light = true; }
  void switchOff() { light = false; }
  boolean isOn() { return light; }
}

LightBulb lb = new LightBulb();
lb.light = true; // Erlaubt!
println("Lampe ist an: " + lb.light);
// Ausgabe: Lampe ist an: true
```

Sketch-Datei und Methoden (1)

- Obwohl ein Sketch-Programm eine Klasse definiert, darf diese nicht ohne weiteres Methoden enthalten
- Folgendes ist nicht erlaubt in einer Sketch-Datei und führt zu einer Fehlermeldung:

```
double a3=9.0, a2=8.0, a1=7.0, a0=6.0, x=3.0;

double polynomEvaluation(double pa3,
                         double pa2, double pa1,
                         double pa0, double px) {
  return(((pa3*px + pa2)*px + pa1)*px + pa0);
}

println("p(" + x + ") = " +
        polynomEvaluation(a3, a2, a1, a0, x));
```

Sketch-Datei und Methoden (2)

- Um Methoden in einer Sketch-Datei verwenden zu können, benötigt diese einen „Konstruktor“ in Form der `setup()`-Methode
- Diese Methode dient als Konstruktor der Programm-Klasse und wird automatisch beim Programmstart aufgerufen
- Das Hauptprogramm wird dann in die `setup()`-Methode geschrieben
- Sobald eine `setup()`-Methode vorhanden ist, können weitere Methoden verwendet werden, ohne dass hierfür eigene Klassen erforderlich wären

Sketch-Datei mit setup()-Methode

```
double a3=9.0, a2=8.0, a1=7.0, a0=6.0, x=3.0;

double polynomEvaluation(double pa3, double pa2,
                         double pa1, double pa0,
                         double px) {
  return(((pa3*px + pa2)*px + pa1)*px + pa0);
}


void setup() { // Hauptprogramm jetzt in setup()
  println("p(" + x + ") = " +
      polynomEvaluation(a3, a2, a1, a0, x));
}
```

- Ausgabe: `p(3.0) = 342.0`

Statische eingebettete Klassen

- (Engl.: *Nested classes*)
- Bezeichnet eine statische innere Klasse
- Diese Klassen sind assoziiert mit ihrer umgebenden Klasse, aber **nicht** mit Instanzen dieser Klasse
  - Analog zu statischen Variablen und Methoden
- Hebt die Bindung zwischen Instanzen der inneren und äußeren Klasse auf

Statische eingebettete Klassen

- Konsequenzen:
  - Um eine statische eingebettete Klasse zu erzeugen ist keine Instanz der äußeren Klasse erforderlich
  - Die Komponenten derartiger innerer Klassen können nur auf statische Komponenten der äußeren Klasse zugreifen
  - Sie können statische Komponenten besitzen
    - D. h. statische Variablen, Methoden und Klassen
    - Dies ist für gewöhnliche innere Klassen verboten

Beispiel: Sketch-Datei

```
static class EmbeddedClass {
   void info1() {
       // method1("message 1"); not allowed: compiler error!
       // var1 = 15;              dito
       method2("message 2");  // allowed (static members)
       var2 = 25;
   }
   static void info2() { method2("message 3"); }
}

int var1 = 10;
static int var2 = 20;
void method1 (String message) { println(message); }
static void method2 (String message) { println(message); }

void setup() { // setup erforderlich in diesem Zusammenhang
    // info1() requires an object of type EmbeddedClass
    EmbeddedClass e = new EmbeddedClass();
    e.info1();
    // info2() does not need an object, since its static
    EmbeddedClass.info2();
}
```

Grafikoperationen

- Processing stellt eine Vielzahl an grafischen Operationen zur Verfügung, die sich einfach in einer Sketch mittels der **`setup()`**-Methode benutzen lassen

- Beispiel:

```
void setup() {
  size(200, 200);          // Größe des Grafikfensters (Pixel)
  background(255);         // Hintergrundfarbe: weiß
  stroke(0);               // Linienfarbe: schwarz (grey)
  strokeWeight(2);         // Strichstärke in Pixel
  fill(255,0,0);           // Füllfarbe: rot (R, G, B)
  rect(10, 10, 80, 80); // Rechteck : (x, y, width, height)
  line(10, 10, 90, 90); // Linie zeichnen: (x1, y1, x2, y2)
}
```

- Siehe: https://processing.org/reference/

Vererbung

Einführung

- Vererbung ermöglicht die Definition einer neuen Klasse auf der Grundlage einer schon existierenden
- Dabei erbt die neu definierte Klasse die Datenfelder und Methoden der Oberklasse, aus der sie abgeleitet wurde
- Vererbung ist eine weitere Schlüsseltechnik zur Steigerung der Effizienz in der Programmierung
- Sie ermöglicht, neue Funktionalitäten auf der Grundlage schon existierender Lösungen zu implementieren
- Auf diese Weise wird Code-Duplikation vermieden

Warum Vererbung?

- Oftmals besitzen Klassen ähnliche Eigenschaften
  - D. h. ähnliche Variablen und Methoden

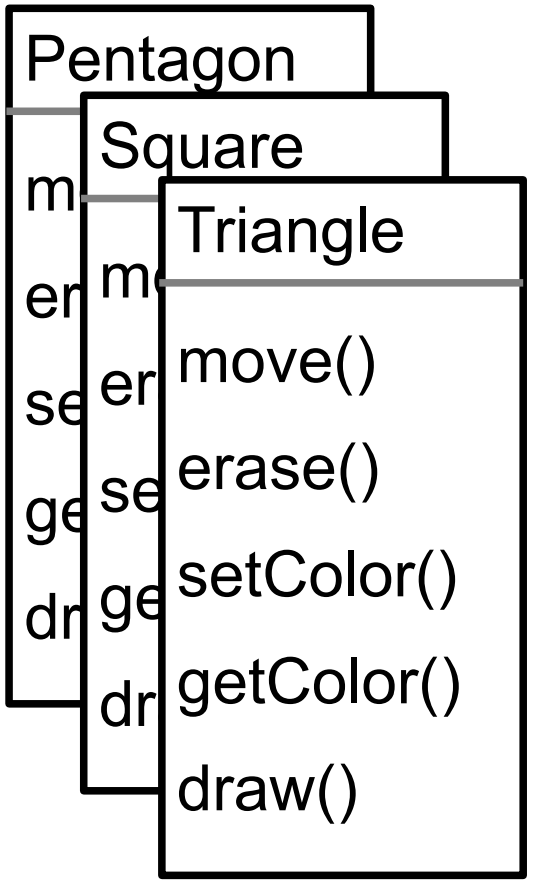
Triangle
move()
erase()
setColor()
getColor()
draw()

Square
move()
erase()
setColor()
getColor()
draw()

Pentagon
move()
erase()
setColor()
getColor()
draw()

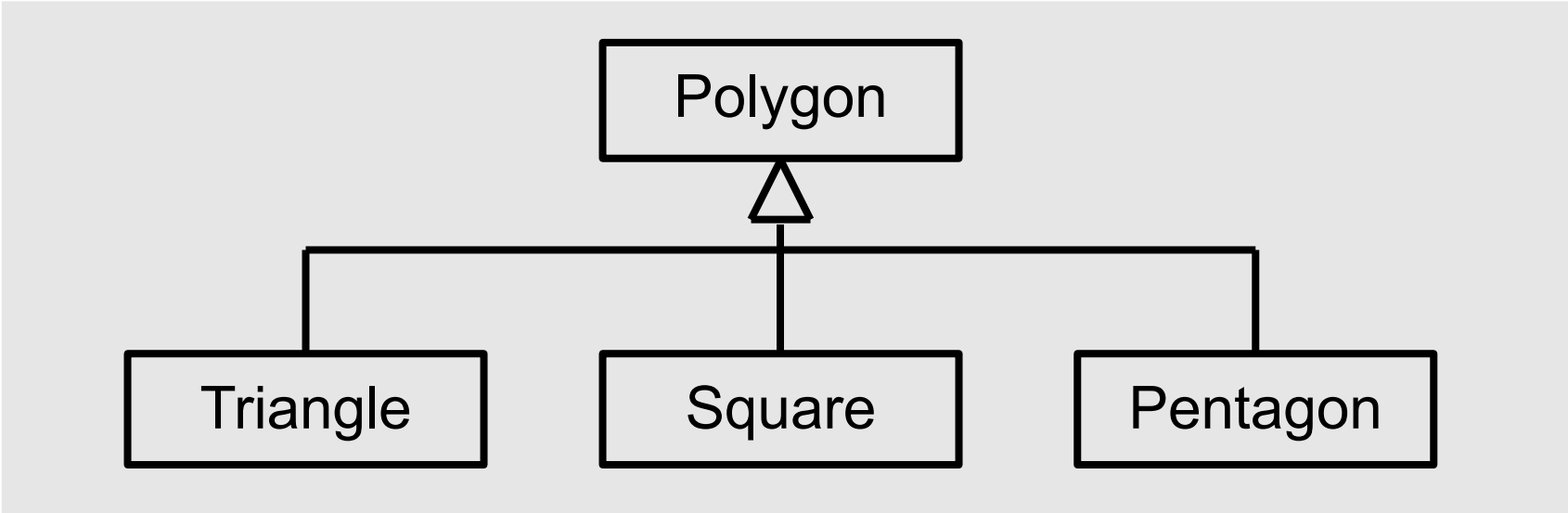
Warum Vererbung?

- Klassen, die nahezu gleiche Eigenschaften aufweisen, mehrfach zu implementieren ist:
  - Zeitverschwendung
  - Eine Quelle möglicher Inkonsistenzen
  - Teuer in der Wartung



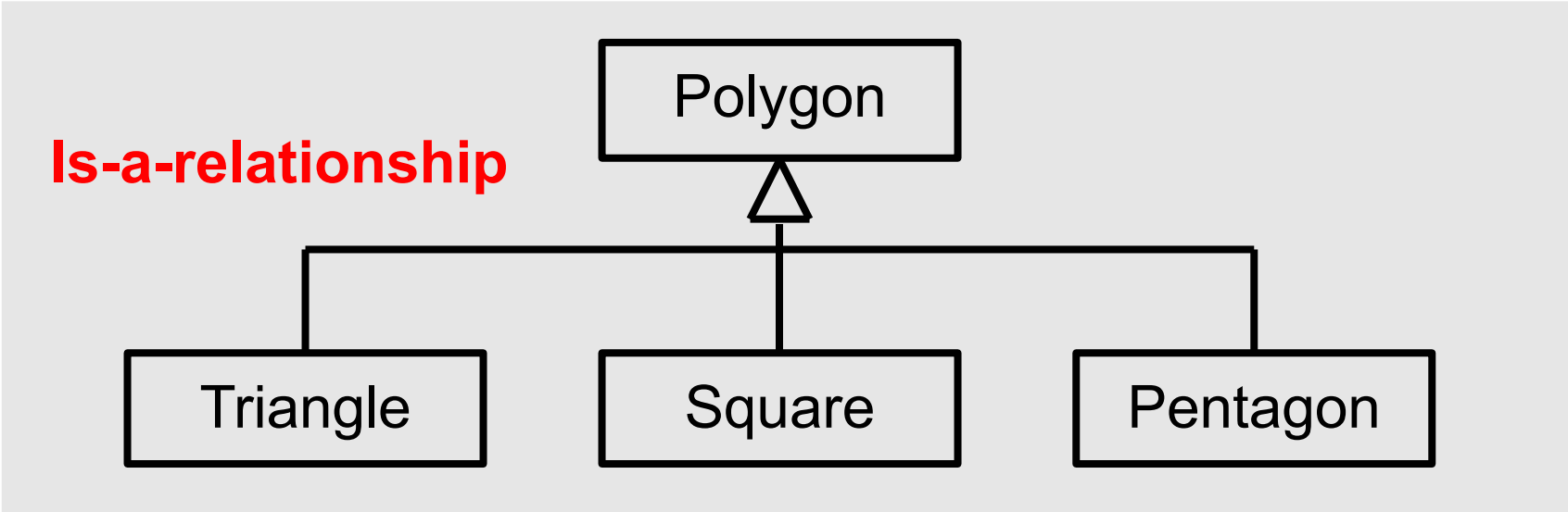
Warum Vererbung?

- Besser: Eine **Oberklasse** (auch: Basisklasse, Superklasse, engl.: *super class*) definieren, die die Funktionalitäten zur Verfügung stellt, die für alle Klassen gleich sind
- Eine Reihe von **Unterklassen** (engl.: *sub classes*) aus dieser Oberklasse ableiten, die die Besonderheiten implementieren



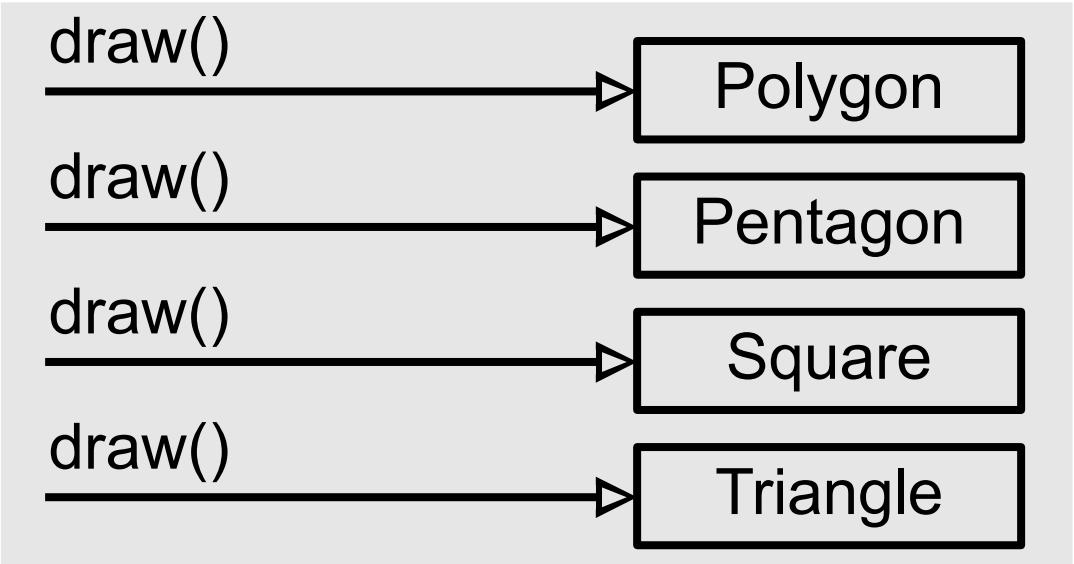
Warum Vererbung?

- Die Unterklassen **erben** (engl.: *inherit*) ihre Schnittstelle von der Oberklasse
- Übergang vom Allgemeinen (Oberklasse) zum Besonderen (Unterklasse)



Schnittstellenkompatibilität

- Der Aufruf einer Methode wird oftmals interpretiert als Senden einer Nachricht an eine Klasseninstanz
- Da Unterklassen die Schnittstelle der Oberklasse erben, können alle Nachrichten, die an die Oberklasse geschickt werden können, auch an die Unterklassen geschickt werden



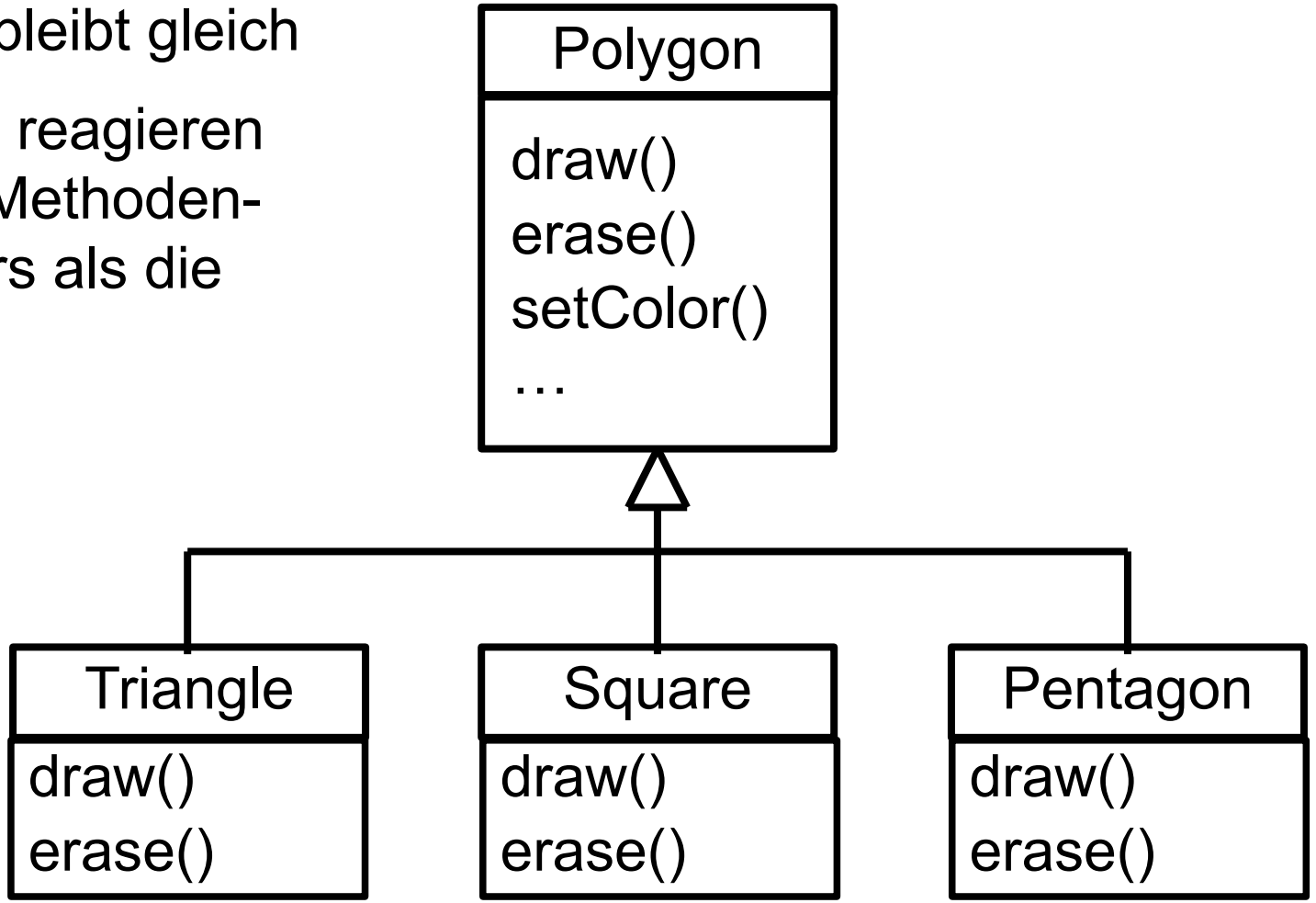
- Die Antwort der Unterklasse kann aber von der der Oberklasse verschieden sein!

Vererbung

- Ohne Abänderung sind die Unterklassen identisch zur Oberklasse
- Wie lassen sich Änderungen durchführen?
  - Durch **Überschreiben** (engl.: *override*) existierender Methoden oder Variablen
    - Ersetzungsprinzip
  - Durch **Überladen** (engl.: *overloading*) und **Hinzufügen** neuer Methoden oder Variablen
    - Erweiterungsprinzip
    - Unterklassen werden daher auch als **Erweiterungsklassen** bezeichnet
  - Das Überschreiben, Überladen oder Hinzufügen von Methoden führt zu Änderungen des Verhaltens

Überschreiben von Methoden der Oberklasse

- Schnittstelle bleibt gleich
- Unterklassen reagieren auf manche Methoden-aufrufe anders als die Oberklasse

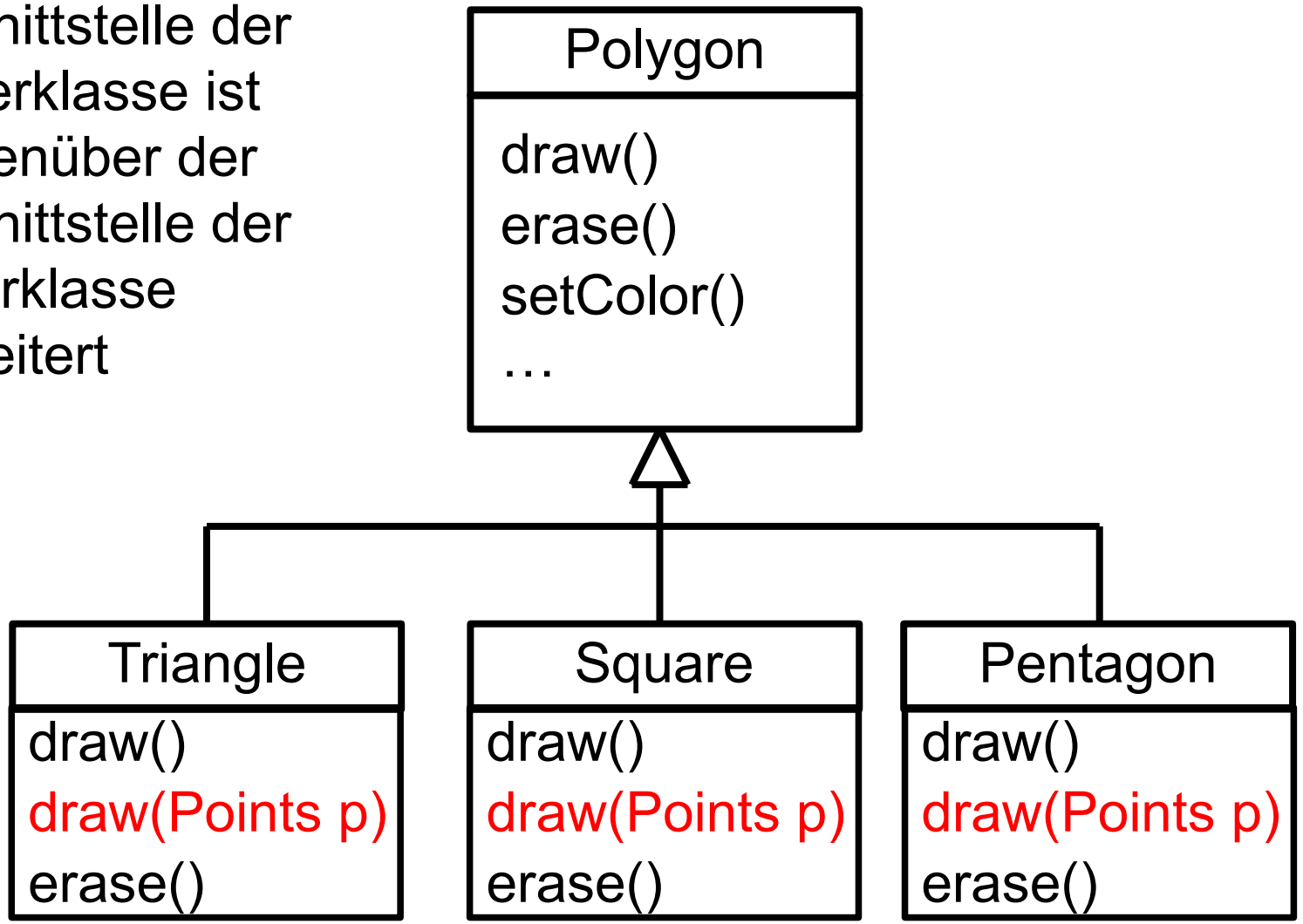


Überschreiben von Methoden: Regeln

- Die überschreibende Methode muss die gleiche **Signatur** (engl.: *signature*) wie die Basisklassenmethode aufweisen
- Zur Signatur gehören:
  - Der **Methodenname**
  - Die **Methodenparameter**
- Bei gleicher Signatur müssen der Methodenname sowie die Anzahl und die Datentypen der Methodenparameter übereinstimmen
  - Die Namen der Parameter in der überschreibenden Methode müssen jedoch nicht mit den Namen der Parameter in der Basisklasse übereinstimmen
- Der Typ des Rückgabewerts gehört nicht zur Signatur!

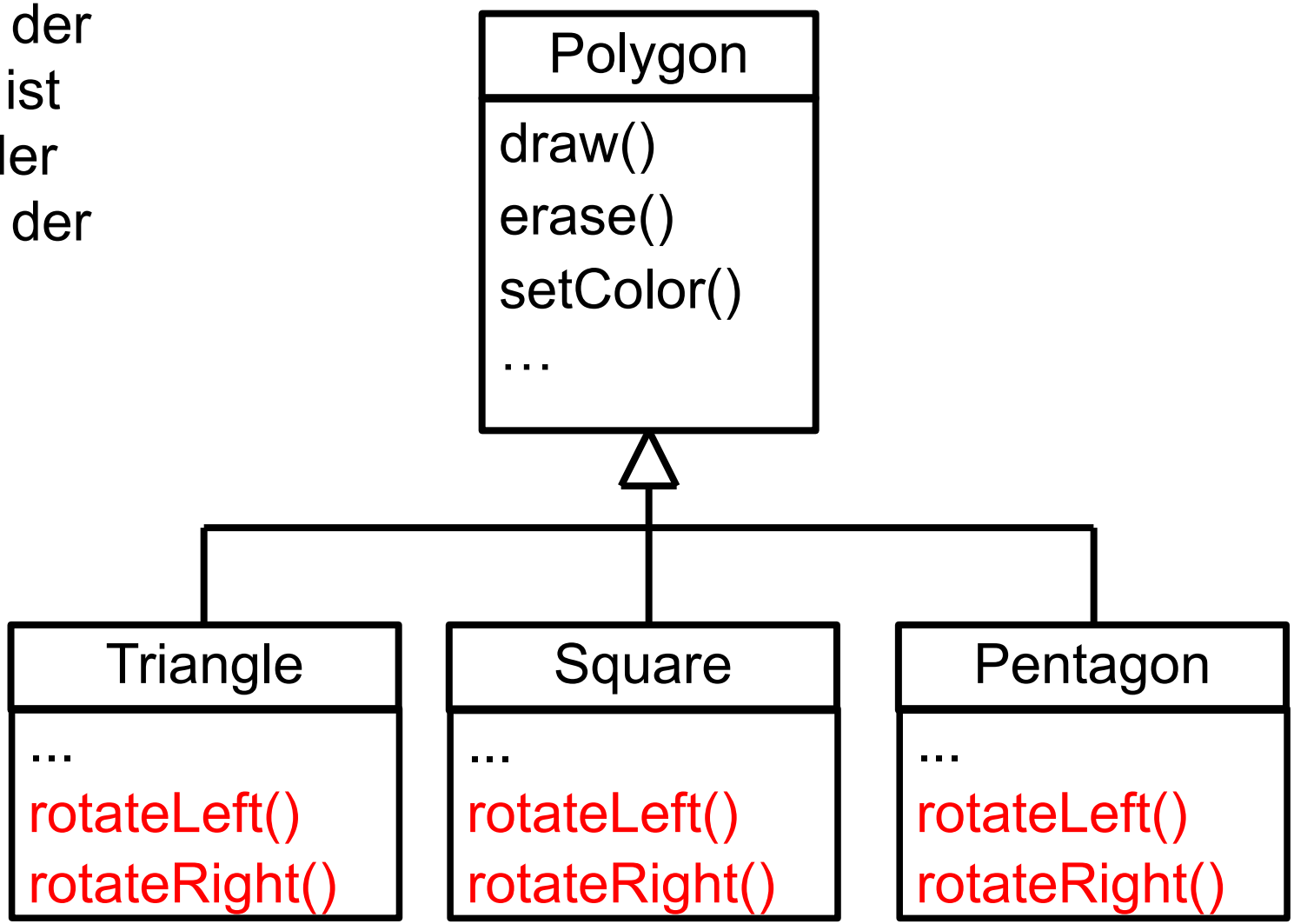
Überladen von Methoden der Oberklasse

- Schnittstelle der Unterklasse ist gegenüber der Schnittstelle der Oberklasse erweitert



Hinzufügen von Methoden

- Schnittstelle der Unterklasse ist gegenüber der Schnittstelle der Oberklasse erweitert



Vererbung in Java

- Wird durch Angabe des Schlüsselworts **`extends`** eingeleitet

```
class Triangle extends Polygon {

    ...

}
```

- In Java erben alle Unterklassen nur von **einer** Oberklasse
- Dies wird als **single inheritance** bezeichnet
  - C++ z. B. bietet *multiple inheritance*

Beispiel

```
public class Polygon { // Super class

   public void setColor(Color c) {
     // set color
   }

   public void draw( /*parameter*/ ) {
     // draw polygon
   }
}

public class Triangle extends Polygon {

    // use setColor from super class

    public void draw( /*parameter*/ ) {
      // draw triangle
    }
}
```

Schlüsselwort `super`

- Aufruf des Konstruktors der Oberklasse im Konstruktor der Unterklasse
  - `super(<params>);`
  - Delegation der Verarbeitung bestimmter Parameterwerte an den Konstruktor der Oberklasse
  - **`super`** muss erster Befehl im Konstruktorrumpf sein
- Ist der Konstruktor der Oberklasse überladen, so wird derjenige mit passender Signatur (d. h. Parameterliste) ausgewählt
  - Entsprechend der Behandlung überladener Methoden

Zusammenspiel von Ober- und Unterklasse

- Ob Aufrufe von **`super`** in der Unterklasse erforderlich sind, ist vom Aufbau der Oberklasse abhängig:

Oberklasse	Unterklasse
Ohne Konstruktor	Kein **`super`**-Aufruf erforderlich
Konstruktor ohne Parameter	Kein **`super`**-Aufruf erforderlich
Konstruktor mit Parameter	Eigener Konstruktor mit **`super`**-Aufruf erforderlich. Der eigene Konstruktor muss mindestens die Parameterwerte für den Oberklassenkonstruktor einsammeln

- Der **`super`**-Aufruf muss vorhanden sein, wenn der Konstruktor der Oberklasse Parameter besitzt

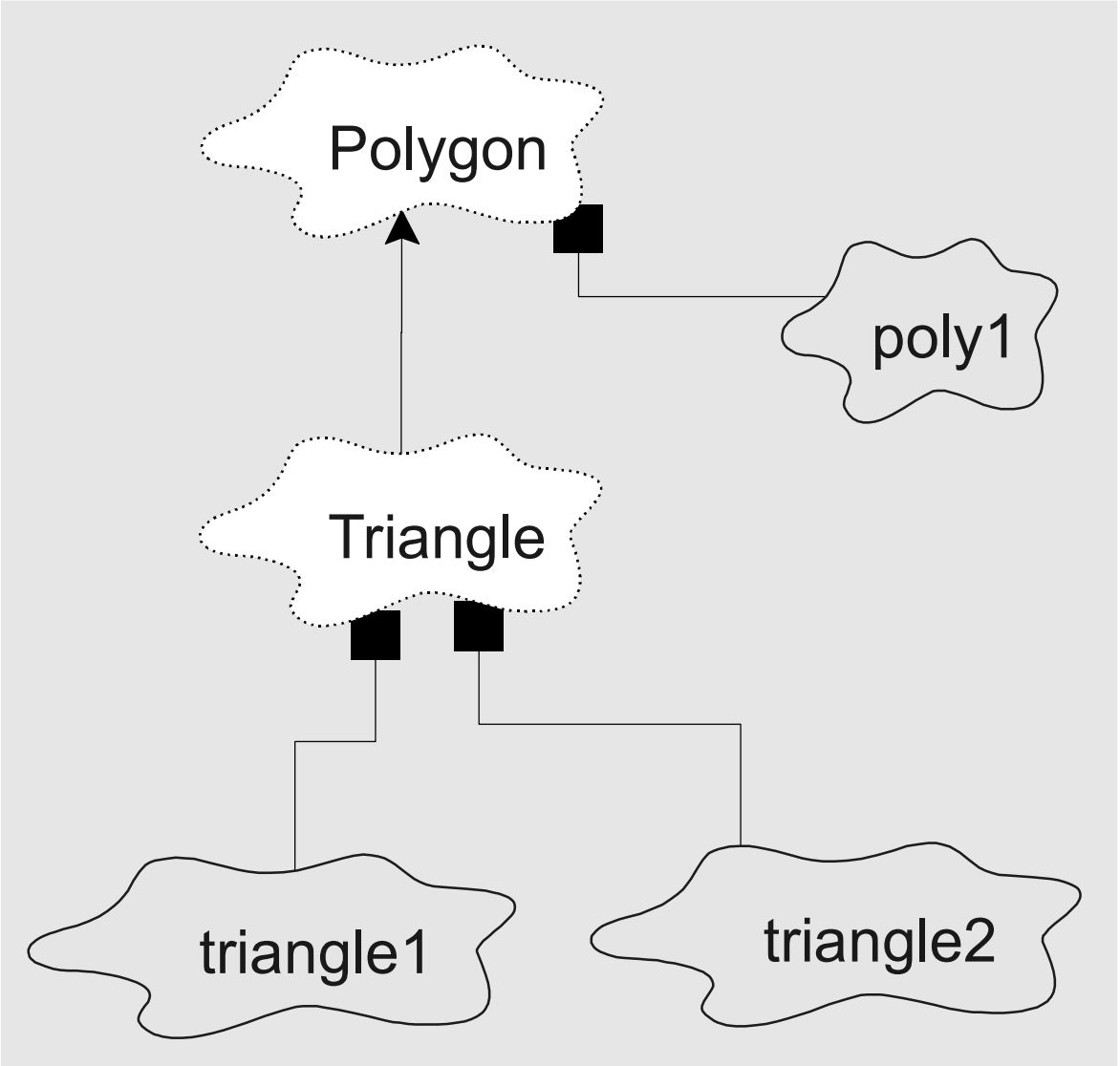
Schlüsselwort `super`

- Aufruf von *gleichnamiger* Methode der Oberklasse mit Hilfe von **`super`** und Punkt-Operator:
  - `super.<methodName> (<params>);`
  - Beispiel: `super.meineMethode(pVal);`
- Ermöglicht Vermeidung von Coderedundanz
- Auch hier gilt: Ist die Methode überladen, so wird die Variante mit passender Signatur ausgewählt
- Zugriff auf *gleichnamige* Variable der Oberklasse:
  - `super.<Variablenname>;`
  - Beispiel: `super.myVar;`

Bsp.: Aufruf des Konstruktors der Oberklasse

```
class Polygon { // Super class
  private Color c;
  Polygon(Color c) {
    this.c = c;
  }
  public void draw( /*parameter*/ ) { /*draw polygon*/ }
}
class Triangle extends Polygon {
  int size;
  Triangle(Color c, int size) {
    super(c); // must be first statement in body
    this.size = size;
  }
  public void draw( /*parameter*/ ){ /*draw triangle*/ }
}
```

Beispiel



- Vererbung findet zwischen Klassen, aber nicht zwischen Objekten statt!

Beispiel: Medienbibliothek

- Klasse für die Repräsentation von allgemeinen Medien

```
class Medium {

   String title;  // name of item
   String format; // mp3, pdf, mpg,...

   // Konstruktoren
   Medium(String title, String format) {
      this.title = title;
      this.format = format;
   }

   Medium() {
      this.title = "no";
      this.format = "no";
   }
```

- Der zweite Konstruktor ermöglicht den Fall
  `Medium m = new Medium();`
- Anderenfalls müssten bei der Instanzierung zwingend Parameterwerte angegeben werden

Klasse Medium: Ausgabe der Daten

```
  void show() {
     System.out.println(title + ", " + format);
   }
}
```

Sketch-Hauptprogramm

- Abspeichern und Anzeigen von Informationen über Mediendateien

```
Medium m1 = new Medium("A Song", "mp3");
Medium m2 = new Medium("A Book", "pdf");
Medium m3 = new Medium("A Movie", "mpg");

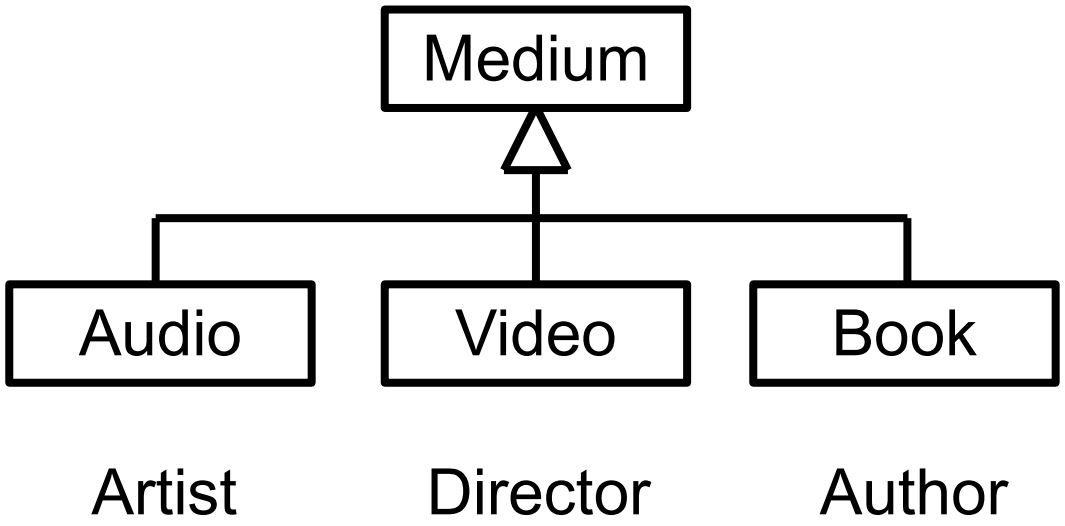
m1.show();
m2.show();
m3.show();
```

- Ausgabe:

```
A Song, mp3
A Book, pdf
A Movie, mpg
```

Anwendung von Vererbung

- Um spezifische Informationen zu unterschiedlichen Medienformen abspeichern zu können, definieren wir die Unterklassen **`Audio`**, **`Video`** und **`Book`**



- Motivation: Klassen durch Vererbung "schlank" halten

Anwendung von Vererbung

```
class Audio extends Medium {

  String artist; // Neu: Musikername

  Audio(String artist, String title, String format) {
    super(title, format); // erste Anweisung im Rumpf
    this.artist = artist;
  }

  Audio() {
    // super(); hier nicht
    // erforderlich. Der Aufruf
    // erfolgt im parameterlosen
    // Fall automatisch
    this.artist = "no";
  }
}
```

- Der Aufruf des Konstruktors der Oberklasse muss die erste Anweisung im Konstruktorrumpf sein
- Definitionen der Klassen **`Video`** und **`Book`** entsprechend

Neues Sketch-Hauptprogramm

```
Audio a1 = new Audio("An Artist", "A Song", "mp3");
Medium m2 = new Medium("A Book", "pdf");
Medium m3 = new Medium("A Movie", "webm");
a1.show();
m2.show();
m3.show();
```

- Ausgabe:

```
A Song, mp3
A Book, pdf
A Movie, webm
```

- Die Klasse **`Audio`** nutzt bislang die **`show()`**-Methode der Klasse **`Medium`**
- Die neue, in Audio abgelegte Information wird nicht ausgegeben!

Vererben mit Überschreiben

- Ausgabe der Zusatzinformationen mittels neuer **`show()`**-Methode
- **`show()`**-Methode überschreibt die gleichnamige Methode der Klasse **`Medium`**

```
void show() {
  System.out.println(artist + ", " + title +
                     ", " + format);
}
```

- Ausgabe im Hauptprogramm:

```
An Artist, A Song, mp3
A Book, pdf
A Movie, webm
```

Redundanzvermeidung

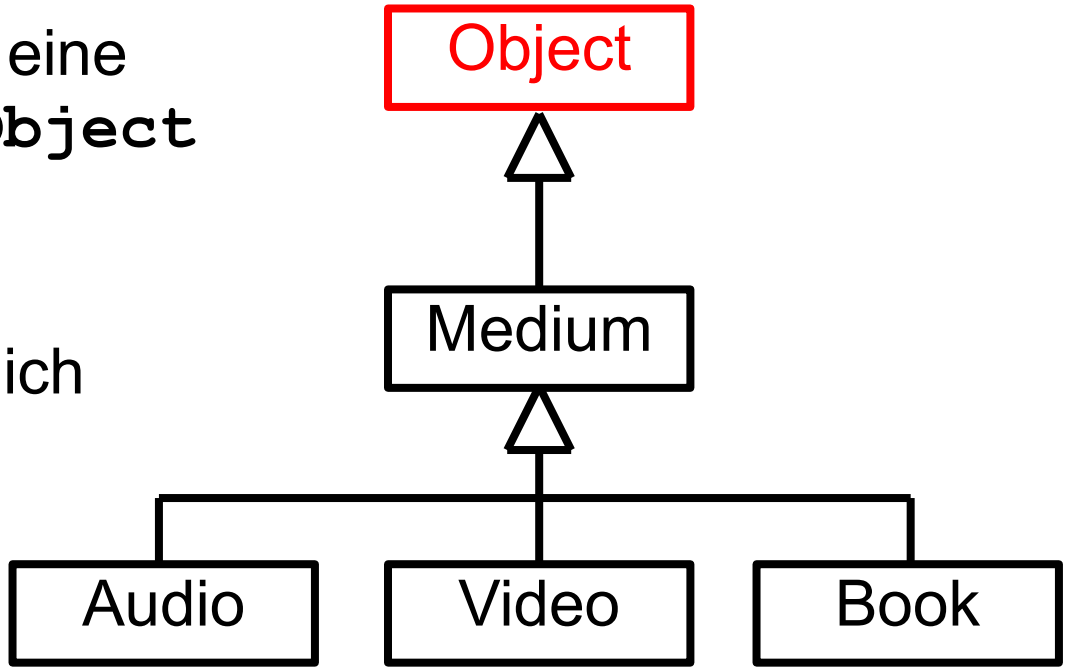
- Ausgabeanweisung in **`show()`** repliziert Code der Oberklasse
- Dies lässt sich durch Aufruf der Methode der Oberklasse vermeiden

```
void show() {
  System.out.println(artist + ", " + title +
                     ", " + format);
}
```

```
void show() {
  System.out.print(artist + ", ");
  super.show();
}
```

`Object`-Klasse

- Alle Klassen in Java haben eine gemeinsame Oberklasse: **`Object`**
- Die Ableitung aus **`Object`** erfolgt implizit, d. h. ohne dass ein **`extends`** erforderlich wäre
- Alle Klassen, die keine andere Klasse erweitern, besitzen automatisch **`Object`** als Oberklasse
- Siehe: *Java API description*



Attribute

Attribute

- Bisher haben wir die folgenden Attribute, die den Zugriff auf Klassenvariablen und -methoden steuern, kennengelernt:
- **`public`**
  - Diese Klassenkomponente darf jeder benutzen
- **`private`**
  - Diese Komponente darf nur innerhalb der Klasse benutzt werden
- **`static`**
  - Diese Komponente gehört zur Klasse, aber NICHT zum Objekt bzw. zur Klasseninstanz

Attribute

- "friendly"
  - "friendly" bezeichnet eine Komponente ohne Attribut
  - Diese Komponente darf innerhalb desselben Pakets (engl.: *package*) benutzt werden
- Daneben gibt es ein weiteres Attribut, welches den Zugriff innerhalb einer Vererbungshierarchie steuert:
- **`protected`**
  - Auf diese Komponente darf nur innerhalb der gleichen Klasse oder des gleichen Pakets zugegriffen werden oder **aus abgeleiteten Klassen**

`final`-Attribut

- Bisher benutzten wir das Attribut **`final`** um Konstanten zu deklarieren
- Hinsichtlich der Vererbung besitzt **`final`** ebenfalls eine Bedeutung
- **`final`** als Teil einer Klassendeklaration
  - Beispiel: **`final class Dinosaur {...}`**
  - Von dieser Klasse darf keine Unterklasse abgeleitet werden

`final`-Attribut

- `final` als Teil einer Methodendeklaration
  - Beispiel:

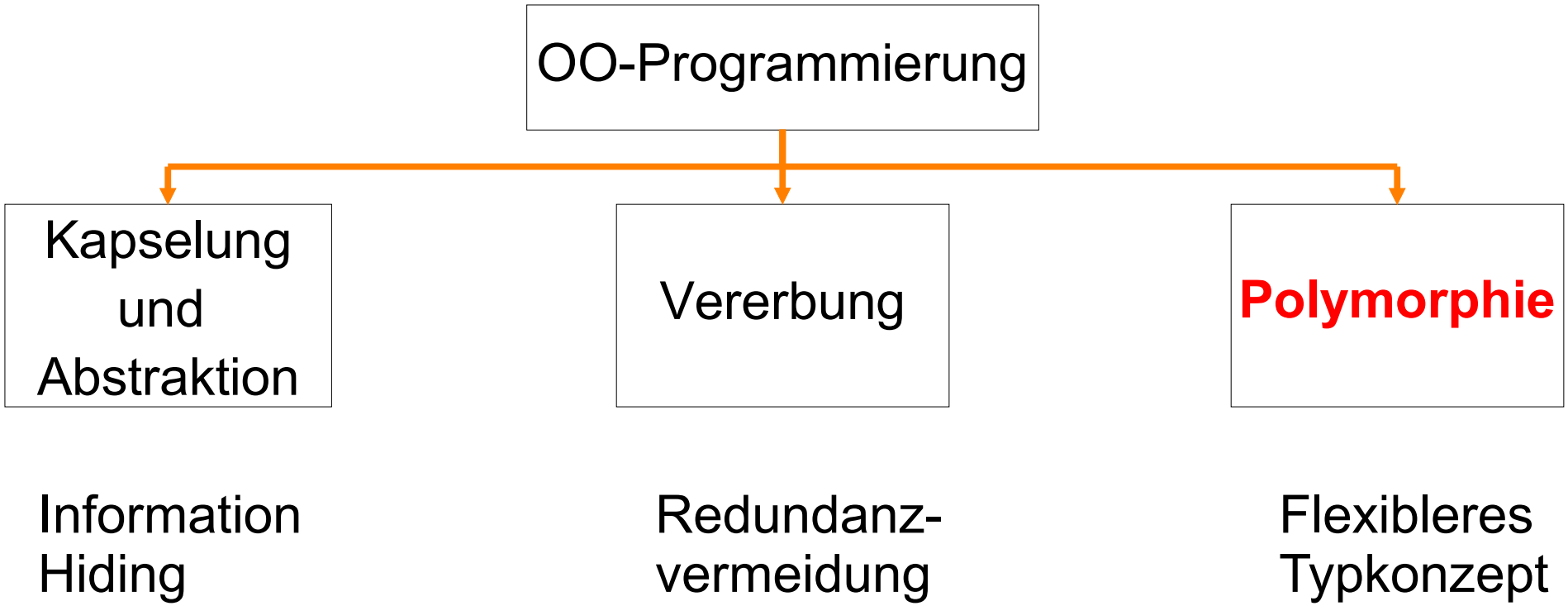
    ```
    final void alwaysTheSame (int param) {...};
    ```

  - Diese Methode darf nicht in einer Unterklasse überschrieben werden
  - Auf diese Weise lässt sich erzwingen, dass eine Methode mit bestimmter Signatur innerhalb des Vererbungsbaumes immer das gleiche tut
  - Überladen mit abweichender Signatur ist hingegen möglich

Polymorphie

Einführung

- Begriff kommt aus dem Griechischen und meint Vielgestaltigkeit (engl.: *polymorphism*)
- Das dritte Standbein der objektorientierten Programmierung

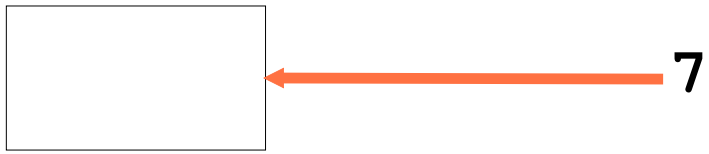


Typprüfung

- Variablen dürfen immer nur Werte oder Objekte des gleichen Typs zugewiesen bekommen
  - Ggf. muss der Typ des Werts oder Objekts vorher umgewandelt werden

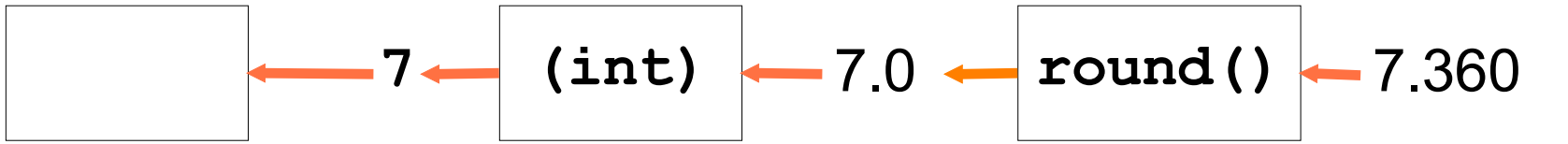
1.

```
int var;
```



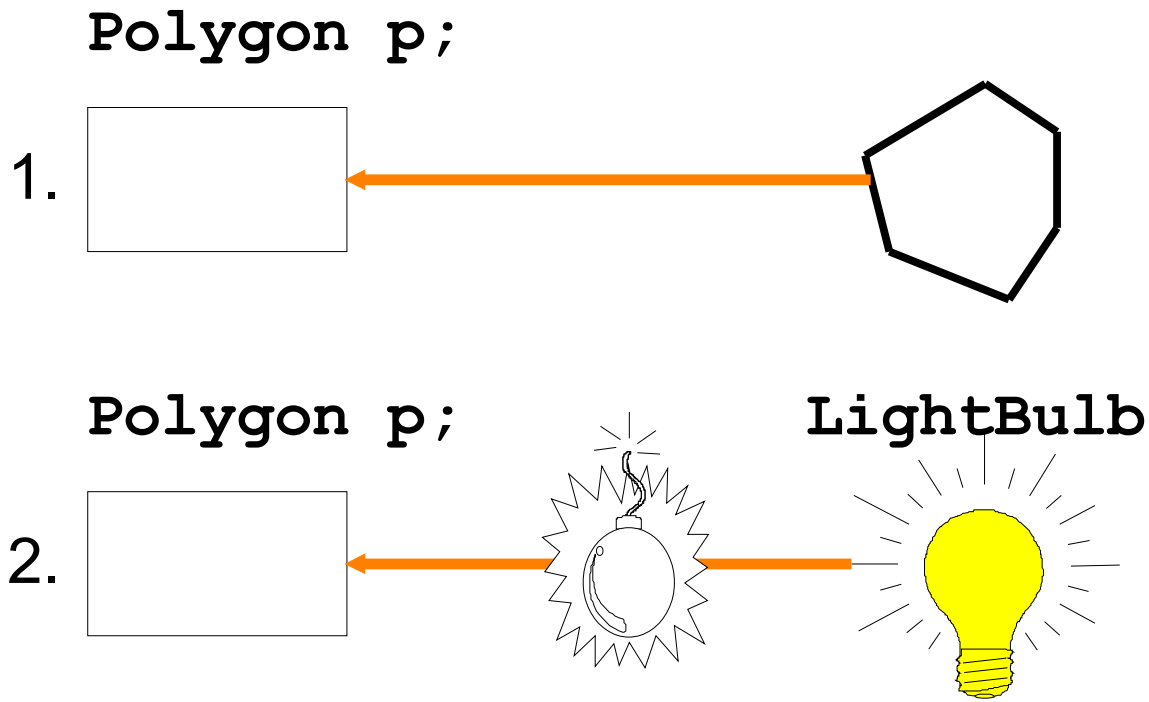
2.

```
int var;
```



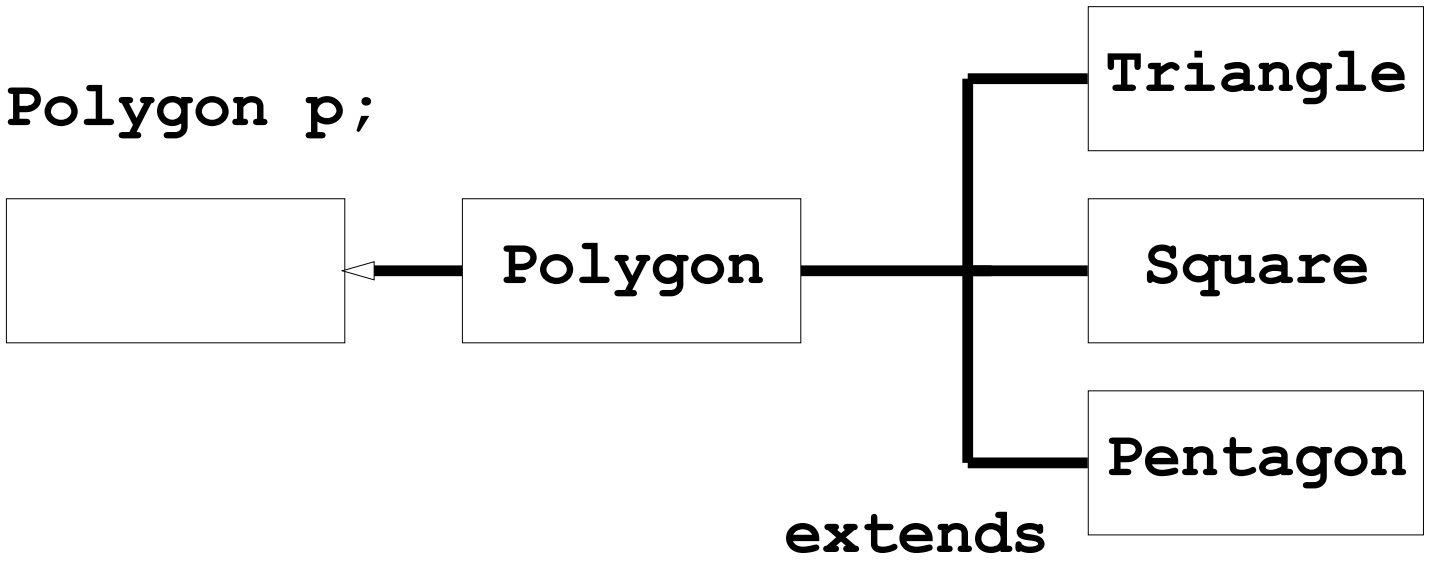
Typprüfung

- Die Beziehung zwischen Variablen- bzw. Parametertypen auf der einen Seite und Wert-Typen auf der anderen Seite sind statisch und schon zur Kompilierzeit bekannt
- Unverträglichkeiten liefern Compilerfehler
- Dies heißt **statische oder frühe Bindung**
- (engl.: ***early binding***)

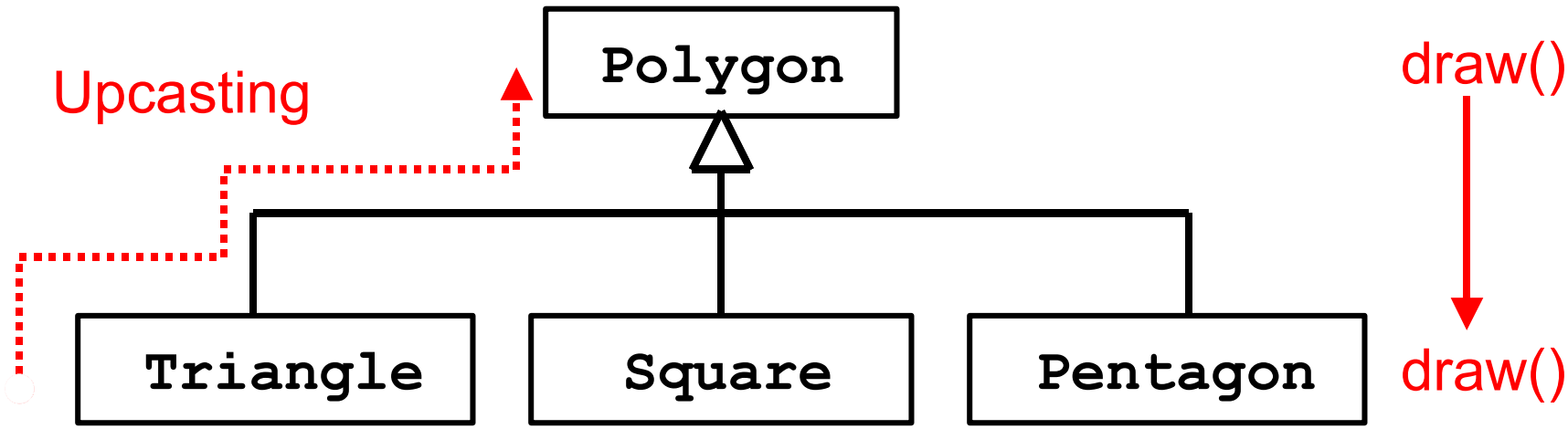


Polymorphie

- Polymorphie beruht auf dem Vererbungskonzept
- Hinsichtlich ihres Typs sind abgeleitete Klassen kompatibel zu ihrer Oberklasse!
- Der Variablen `p` im Beispiel unten können deshalb Objekte unterschiedlichen Typs zugewiesen werden

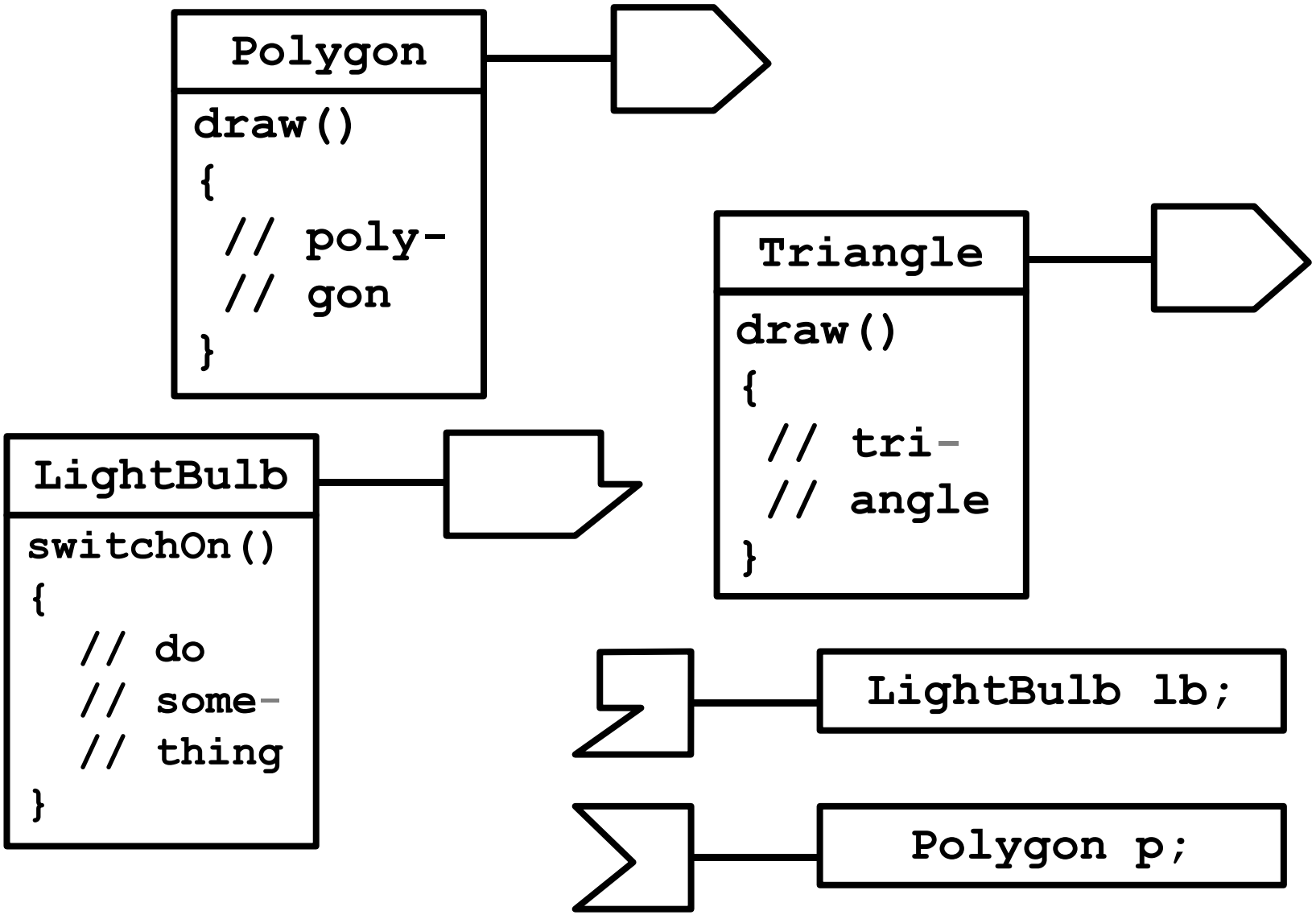


Polymorphie



- Da Objekte während der Laufzeit dynamisch generiert werden können, wird erst zur Laufzeit festgestellt, welchen Typ das Objekt besitzt, auf welches die Referenzvariable der Oberklasse verweist
- Dies wird als **späte Bindung** (engl.: *late binding*, *runtime binding*, *dynamic binding*) bezeichnet

Polymorphie



Beispiel

```
// Polymorphie: Einer Variable können Objekte
// unterschiedlichen Typs zugewiesen werden

Polygon p = new Polygon(pColor);

p = new Triangle(pColor, pSize);
p.draw(x1, y1); // draws a triangle at x1, y1

p = new Square(pColor, pSize);
p.draw(x2, y2); // draws a square at x2, y2

p = new Pentagon(pColor, pSize);
p.draw(x3, y3); // draws a pentagon at x3, y3
```

Polymorphie: Regeln

- Eine Referenzvariable vom Typ der Oberklasse kann Objekte der
  - Oberklasse sowie
  - ihrer Unterklassen bzw. der Unterklassen dieser Unterklassen usw.

  referenzieren

```
Polygon p = new Triangle(pColor, pSize); // erlaubt!!
```

- Eine Referenzvariable vom Typ der Unterklasse kann hingegen kein Objekt der Oberklasse referenzieren!

```
Triangle t = new Polygon(pColor); // NICHT erlaubt!!
```

Upcasting gilt auch für Felder

```
// Polymorphie: Dasselbe Feld kann Objekte
// unterschiedlichen Typs zugewiesen bekommen
Polygon[] polygons = new Polygon[3];

polygons[0] = new Triangle(pColor, pSize);
polygons[1] = new Square(pColor, pSize);
polygons[2] = new Pentagon(pColor, pSize);

polygons[0].draw(x1,y1);   // draws a triangle
polygons[1].draw(x2,y2);   // draws a square
polygons[2].draw(x3,y3);   // draws a pentagon
```

Upcasting gilt auch für Felder

```
Polygon[] polygons = new Polygon[3];

polygons[0] = new Triangle(pColor, pSize);
polygons[1] = new Square(pColor, pSize);
polygons[2] = new Pentagon(pColor, pSize);

int x = 50, y = 50, ydist = 50;

// Ausgabeanweisungen durch Schleife abkürzen
for (int i=0; i<3; i++, y+=ydist)
  polygons[i].draw(x, y);
```

Klasse eines Objekts herausfinden

- Als Folge der Polymorphie können Referenzvariablen zu unterschiedlichen Zeitpunkten Referenzen auf Objekte unterschiedlicher Klassen enthalten
- Code kann aber ggf. von der Klasse des referenzierten Objekts abhängig sein
  - Über das Erweiterungsprinzip können Objekte unterschiedliche erweiterte Funktionalitäten bieten
- Wie läßt sich herausfinden, welche Klasse zu einem Objekt gehört?
- Hierfür besitzt Java den `instanceof`-Operator

Beispiel: Objekte zufällig erzeugen

```
double shape = Math.random(); // Zufallszahl [0; 1]

Polygon p;

if (shape < 0.333)
  p = new Triangle(pColor, pSize);
else if (shape > 0.666)
  p = new Square(pColor, pSize);
else
  p = new Pentagon(pColor, pSize);

// p might now refer a triangle, square or pentagon
```

Beispiel: Objekt zufällig erzeugen

```
if (p instanceof Triangle)
  System.out.println("Triangle generated");
else if (p instanceof Square)
  System.out.println("Square generated");
else
  System.out.println("Pentagon generated");

p.draw(x, y);
```

- **Syntax:** `Objectname instanceof Classname`
- `instanceof`-Ausdruck **liefert** `true` **oder** `false`

Abstrakte Klassen

Abstrakte Klassen

- Oftmals besitzen Oberklassen mehr den Charakter einer Schnittstellendefinition
- D. h. eines Bauplanes, der angibt, über welche Methoden eine Gruppe ähnlicher Klassen verfügen soll
- In dem Grafikbeispiel ist es z. B. sinnvoll, wenn die Klasse **`Polygon`** und alle ihre Unterklassen eine **`draw()`**-Methode implementieren
- Anderenfalls wäre die jeweilige Klasse sinnlos
  - Sie könnte das entsprechende Grafikobjekt nicht auf den Bildschirm zeichnen

Schnittstelle durch "Hilfsklasse"

- Wir könnten eine Klasse **`Shape`** einführen, die eine Methode **`draw(...)`** definiert

```
class Shape {
  public void draw(int x, int y) {
  }
}
```

- Die Methode **`draw(...)`** soll als Signatur (engl.: *signature*) für andere Klassen dienen
  - Zur Signatur gehören der Methodenname und die Typen in der Parameterliste

Schnittstelle durch "Hilfsklasse"

- Da die Methode nur als Signatur dient, hat **`draw(...)`** nichts zu tun

  => Wir lassen den Methodenrumpf leer

```
class Shape {
  public void draw(int x, int y) {
  }
}
```

Schnittstelle durch "Hilfsklasse"

```
class Polygon extends Shape {
  protected int size;
  public Polygon(int size) {
    this.size = size;
  }
  public void draw(int x, int y) {
    System.out.println("Polygon: " + x + ", " + y);
  }
}
```

Schnittstelle durch "Hilfsklasse"

```
class Polygon extends Shape {
  protected int size;
  public Polygon(int size) {
    this.size = size;
  }
}
```

- **Problem: Diese Variante von `Polygon` OHNE Implementierung von `draw(...)` ist auch möglich!!**

Schnittstelle durch "Hilfsklasse"

- Dieser Ansatz hat Nachteile:
  - Die Programmierer werden nicht gezwungen, sich an die Signatur zu halten
  - Wird die Methode der Oberklasse in der Unterklasse nicht überladen, so entsteht daraus kein Fehler!
    - Im Falle eines Aufrufs wird automatisch die Methode der Oberklasse aufgerufen
    - Diese tut aber nichts oder gibt bestenfalls eine Fehlermeldung aus!

Schnittstelle durch "Hilfsklasse"

- Hieraus folgt, dass dieser Fehler entweder gar nicht oder erst während der Laufzeit erkannt wird!
- Es ist immer besser die Entstehung von Fehlern frühzeitig, d. h. schon beim Kompilieren abzufangen!

Abstrakte Klassen

- Eine Schnittstelle, deren Benutzung freiwillig ist, stellt daher eine potentielle Fehlerquelle dar!
- Java bietet deshalb die Möglichkeit, Schnittstellen zu definieren, die implementiert werden müssen
- Dies wird mit Hilfe abstrakter Klassen durchgeführt

Schnittstellendefinition durch abstrakte Klassen

- Abstrakte Klassen werden durch das Schlüsselwort **`abstract`** deklariert

```
abstract class Shape {
  public abstract void draw(int x, int y);
}
```

- Abstrakte Methoden besitzen nur einen Methodenkopf
  - Dieser definiert die Signatur der Methode
- Abstrakte Klassen können nicht instanziert werden!

Beispiel

```
class Polygon extends Shape {
  protected int size;
  public Polygon(int size) {
    this.size = size;
  }
  public void draw(int x, int y) {
    System.out.println("Polygon: " + x + ", " + y);
  }
} // OK! Abstrakte Methode wurde implementiert.
```

Beispiel

```
class Polygon extends Shape {
  protected int size;
  public Polygon(int size) {
    this.size = size;
  }
  public void draw(float x, float y) {
    System.out.println("Polygon: " + x + ", " + y);
  }
}
```

- **Compiler-Fehler: Falsche Parametertypen!**

Beispiel

```
class Polygon extends Shape {
  protected int size;
  public Polygon(int size) {
    this.size = size;
  }
}
```

- **Compiler-Fehler: `draw()` wurde nicht implementiert!**

Regeln

- Nicht alle Methoden einer abstrakten Klasse müssen abstrakt sein
  - Die nicht abstrakten Methoden werden in der üblichen Weise vererbt
  - Abstrakte Methoden müssen in Unterklassen implementiert oder als abstrakt übernommen werden
    - Letzteres geschieht, indem die Unterklasse als **`abstract`** deklariert wird

Regeln

- Sobald irgendeine Methode einer Klasse als **`abstract`** deklariert wird, muss die zugehörende Klasse ebenfalls als **`abstract`** deklariert werden
- Die Definition einer derartigen Klasse ist also nicht abgeschlossen und muss von entsprechenden Unterklassen komplettiert werden
- Abstrakte Klassen können deshalb nicht instanziert werden!

Interfaces

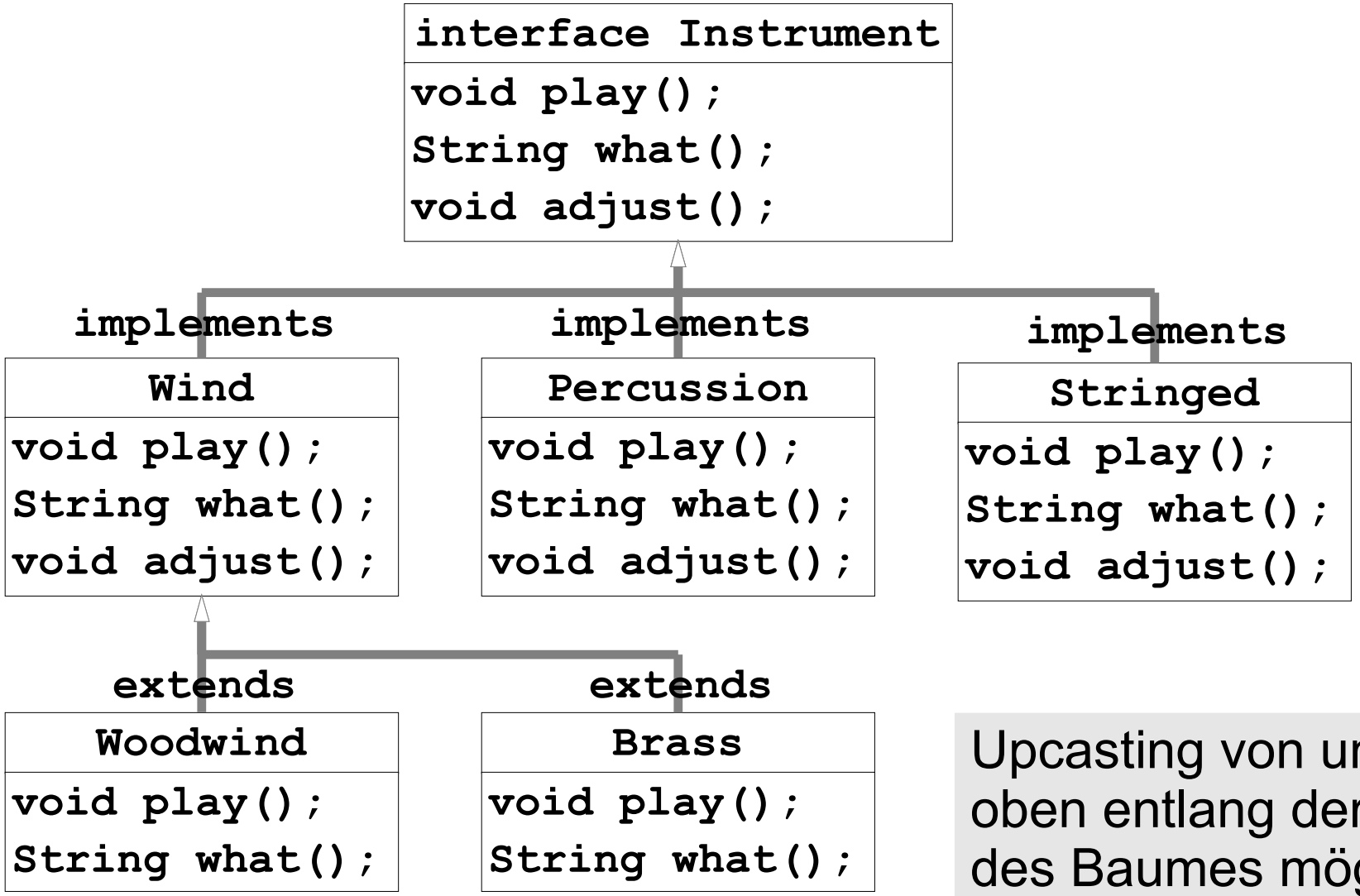
Interfaces

- Mit Hilfe von Interfaces wird das Konzept der abstrakten Klasse hin zu einer reinen Schnittstelle weitergeführt
- In Interfaces werden nur Methodenköpfe deklariert
- Interfaces können keine Methodenrümpfe enthalten
- Interfaces können Datenfelder enthalten, aber nur in Form von Konstanten
- Da ein Interface über die vorgegebenen Methodenköpfe ein bestimmtes Verhalten definiert, spricht man in diesem Zusammenhang auch von einem Protokoll

Interfaces

- Interfaces werden mit Hilfe des Attributs **`interface`** deklariert
  - Dies wird an Stelle des Attributs **`class`** verwendet
- Interfaces werden nicht durch Unterklassen erweitert, sondern von Klassen implementiert
- Hierfür wird das Schlüsselwort **`implements`** benutzt
- Ein Interface entspricht im Prinzip einer abstrakten Klasse, die nur über abstrakte Methoden und Konstanten verfügt

Interfaces



Upcasting von unten nach oben entlang der Kanten des Baumes möglich!

Interface: Beispiel

```
interface Instrument {

  // Compile-time constant:
  static final int I = 5; // static & final:
                          // use as Instrument.I

  // Cannot have method definitions, only headers
  void play(Note n); // Automatically public
  String what();
  void adjust();
}
```

Interface implementieren: Beispiel

```
class Wind implements Instrument {

  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }

  public String what() { return "Wind"; }

  public void adjust() {
    System.out.println("Wind.adjust()");
  }
}
```

Unterklasse ableiten: Beispiel

```
class Woodwind extends Wind {

  public void play(Note n) {

    System.out.println("Woodwind.play() " + n);

  }

  public String what() { return "Woodwind"; }

}
```

Mehrere Interfaces

- Eine Klasse kann auch mehrere Interfaces zur gleichen Zeit implementieren
  - Auf diese Weise kann in Java Mehrfachvererbung simuliert werden
- Es müssen dann alle vorgegebenen Methoden aus allen Interfaces implementiert werden

Mehrere Interfaces

- Die Namen der Interfaces werden durch Kommata voneinander getrennt

- Beispiel:

```
public class MyClass implements InterfaceA,
                                InterfaceB, InterfaceC
{
   // ordinary class definition body
}
```

Interfaces und gewöhnliche Vererbung

- Eine Klasse kann Interfaces implementieren und gleichzeitig eine Oberklasse erweitern

- Beispiel:

```
public class MyClass extends Parent
    implements InterfaceA, InterfaceB, InterfaceC
{
  // ordinary class definition body
}
```

Beispiel

```
interface CanFight {
  void fight();
}

class ActionCharacter {
  public void fight() {}
}
```

```
interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}
```

```
class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
  // fight() already implemented by ActionCharacter!
}
```

Beispiel

```
// Sketch-Hauptprogramm:

void t(CanFight x) { x.fight(); }
void u(CanSwim x) { x.swim(); }
void v(CanFly x) { x.fly(); }

void w(ActionCharacter x) { x.fight(); }


void setup() { // erforderlich für die Methoden oben
  Hero h = new Hero();
  t(h); // Treat it as a CanFight
  u(h); // Treat it as a CanSwim
  v(h); // Treat it as a CanFly
  w(h); // Treat it as an ActionCharacter
}
```

Generische Programmierung

Klasse `Container`

- Erzeugt wird eine Klasse für das Speichern eines Datenwerts
-
```
class Container {
  private int val;
  Container(int val) { this.val = val;}
  void set(int val) { this.val = val; }
  int get() { return val; }
};
```
- Datentyp der Komponenten ist **`int`**
- Die Klasse **`Container`** soll allgemeiner für beliebige Dinge einsetzbar sein
  - Integers, Floats, Lightbulbs, Customers etc.
- Wie kann dies erreicht werden? Lösung: Kopieren mit neuem Namen

Varianten

- ```
  class FContainer {
    private float val;
    Container(float val) { this.val = val;}
    void set(float val) { this.val = val; }
    float get() { return val; }
  };
  ```
- ```
  class LBContainer {
    private LightBulb val;
    Container(LightBulb val) { this.val = val;}
    void set(LightBulb val) { this.val = val; }
    LightBulb get() { return val; }
  };
  ```
- Nachteile: Codeduplikation und unterschiedlicher Code für die gleiche Funktionalität

Generische Programmierung

- Generische Programmierung erlaubt die Parametrierung der Datentypen: Die Datentypen werden zu Variablen
- Beispiel: Verallgemeinerte Containerklasse

```
class Container<T> {
  private T val;
  Container(T val) { this.val = val; }

  void set(T val) { this.val = val; }
  T get() { return val; }
};
```

- Typ-Variablen (hier: `T`) werden zur Kompilierzeit durch die konkret verwendeten Typen ersetzt

Anwendung

- Sketch-Hauptprogramm:

```
Container<Integer> ci = new Container<>(10);
println(ci.get());
Container<Float> cf = new Container<>(11.23);
println(cf.get());
LightBulb lb = new LightBulb();
lb.switchOn();
Container<LightBulb> clb = new Container<>(lb);
LightBulb lb2 = clb.get();
println(lb2.isOn());
```

- Der zu verwendende Datentyp wird in <>-Zeichen notiert
- Es müssen die übergeordneten Klassen zu den primitiven Datentypen eingesetzt werden

Ausgabe:

```
10
11.23
true
```

Generische Programmierung

- Generische Programmierung (engl.: *generic programming*) bezeichnet eine Methode, Algorithmen durch einen weitgehenden Verzicht auf die Festlegung von Datentypen möglichst allgemein und damit abstrakt zu beschreiben
- Die Problemlösungen sind **polymorph** ("vielgestaltig")
  - Bezeichner können abhängig von ihrer Anwendung unterschiedliche Datentypen annehmen
- Dies fördert die Wiederverwendbarkeit der Problemlösungen
- Da diese Vielgestaltigkeit durch den Compiler zur Kompilierzeit aufgelöst wird, spricht man hier auch von **statischer Polymorphie**

Polymorphie: Varianten

- **Statische (Kompilationszeit-) Polymorphie**: Die konkret zu verwendenden Datentypen werden vom Compiler bestimmt
- **Dynamische (Laufzeit-) Polymorphie**: Die konkret auftretenden Datentypen stellen sich erst zur Laufzeit heraus. Die Typprüfung und damit Auswahl passender Funktionen und Objekte findet erst zur Laufzeit statt
- **Überladene** Operatoren und Methoden
  - Die konkrete Funktion ergibt sich aus den Datentypen der Parameter bzw. Operanden

Grafische Bediensysteme

Entwicklungsgeschichte

- Ivan Sutherland
  - Urvater der Computergrafik
- Sketchpad System (1963)
  - Erlaubte Benutzern das interaktive Malen auf dem Bildschirm
- Auf diese Weise wurden Computer für Zeichenaufgaben intuitiv nutzbar
- Hauptmerkmale des Systems:
  - Ein neues Zeigeinstrument, der "light pen“
  - Die Möglichkeit, "Gummibandlinien" zu zeichnen
  - Die Präsentation der Inhalte in Fenstern

Entwicklungsgeschichte

- Konzept wurde in den 1970er Jahren im Palo Alto Research Center von Xerox in Kalifornien weiterentwickelt und um die "*desktop metapher*" ergänzt
- Ziel: Die typischen Arbeitsmittel in einem Büro
  - Schreibmaschine, Terminkalender, Uhr,...
- sollen grafisch auf dem Rechner nachgebildet werden und damit genauso intuitiv wie die Originale benutzbar sein
- Dies wird ebenfalls durch Einführung eines einheitlichen Layouts der Benutzeroberfläche unterstützt

Entwicklungsgeschichte

- Die Interaktion zwischen dem Benutzer und den Programmen erfolgt mit Hilfe der Tastatur und eines neuen Zeigeinstrumentes: Der "Maus"
- Das textorientierte Bediensystem wird durch ein grafisches Bediensystem ersetzt
- Dieses präsentiert sich dem Benutzer in Fenstern
- Jedem Programm ist i. d. R. zumindest ein Fenster zugeordnet
- Der Programmzustand wird im Fenster angezeigt

Entwicklungsgeschichte

- Die Fenster können beliebig auf dem Bildschirm angeordnet werden
- Sie können in der Größe verändert, in den Vordergrund geholt oder ikonisiert, d. h. auf eine minimale Größe verkleinert werden
- Die Steuerung des zugehörenden Programms erfolgt zu einem wesentlichen Teil grafisch mit Hilfe von Menüs und Buttons
- Der Durchbruch dieses Paradigmas gelang 1984 mit Hilfe des Macintosh-Computers der Firma Apple

GUI

- Es wird in diesem Zusammenhang von einer grafischen Bedienoberfläche bzw. Benutzerschnittstelle gesprochen
- Engl.: GUI (Graphical User Interface)
  - Eine grafische Bedienoberfläche wird aus ikonischen Elementen - sog. GUI-Komponenten - zusammengesetzt und in einem Fenster angezeigt
  - GUI-Komponenten werden häufig als Widgets („Window Gadget“) bezeichnet
  - Die Interaktion mit dem Programm erfolgt mit Hilfe der Computermaus durch Auswählen und Drücken dieser Komponenten

GUI-Entwicklung

- Erfordert zunächst geeignete unterlagerte Grafikbibliotheken
- Unter Processing bzw. Java sind die wichtigsten:
  - Java-Widgets (AWT, Swing)
  - G4P-Widgets
    (siehe http://www.lagers.org.uk/g4p/)
- Erfolgt entweder programmatisch oder mit Hilfe eines sogenannten GUI-Builders
  - GUI-Builder: Grafisches Werkzeug mit dem eine GUI mit Zeichenwerkzeugen erstellt wird. Der Programmcode wird automatisch zu dieser Grafik generiert
- Die programmatische Erstellung hat Vorteile im Hinblick auf die heutzutage wichtige Entwicklung dynamischer, responsiver Bedienoberflächen

Java-Programm mit GUI

- Eine Basisanwendung mit Fenster ist bereits durch die Klasse **`JFrame`** implementiert
  - **`JFrame`** ist ein **Container** für die Widgets einer GUI
- Die Klasse des eigenen Programms erbt und erweitert diese Klasse durch eigene Widgets:

```
class DMProgGUI extends JFrame {
   ...
}
```

- Die Klasse kann als innere Klasse in der Sketch-Datei des Hauptprogramms implementiert werden
- Die Widgets werden entsprechend zu den Erfordernissen der eigenen GUI hinzugefügt

Bibliotheken

- Um **`JFrame`** nutzen zu können, müssen die Grafikbibliotheken AWT und Swing importiert werden:

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

class DMProgGUI extends JFrame {
   ...
}
```

Programm mit Fenster erzeugen

- Die GUI muss in der Sketch-Datei von Processing explizit erzeugt werden
- Beispiel:

```
size(400, 600); // Größe des Grafik-Fensters
                // von Processing

DMProgGUI app = new DMProgGUI();
```

- Danach existiert die grafische Bedienoberfläche parallel zum Grafikfenster von Processing
  - Das Bedienfenster der grafischen Bedienoberfläche und das Grafik-Fenster von Processing sind zwei verschiedene Fenster

Bedienfenster konfigurieren

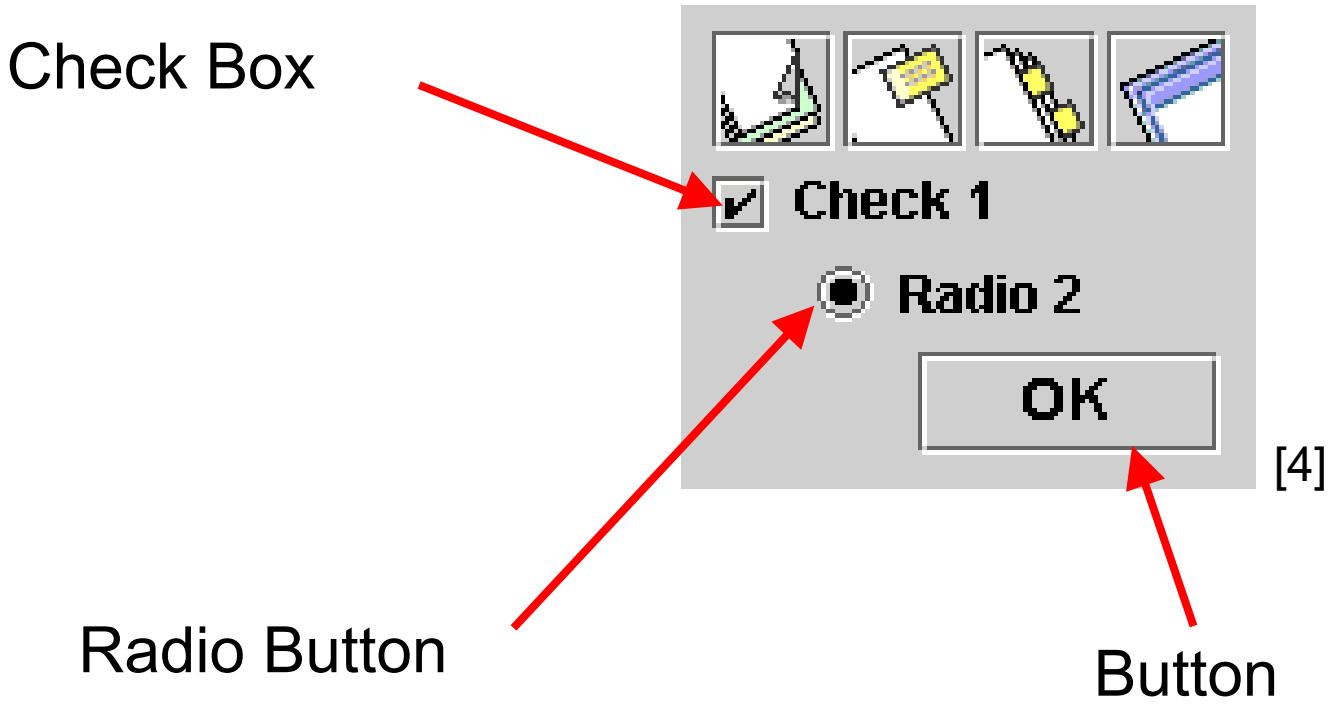
- Die Konfiguration der GUI erfolgt in deren Konstruktor
- In diesem Zusammenhang werden Widgets hinzugefügt

```
DMProgGUI() {
   setTitle("MyAppName");
   setSize(300, 300); // Pixels: width & height
   setVisible(true); // Erforderlich: GUI anzeigen
}
```

  - Das Bedienfenster kann durch Aufruf von `setTitle()` einen Namen erhalten und seine Größe durch Aufruf von `setSize()` in Pixel spezifiziert werden.
  - Die GUI wird nur angezeigt, wenn `setVisible` aufgerufen wurde!

Grundlegende Widgets

Buttons

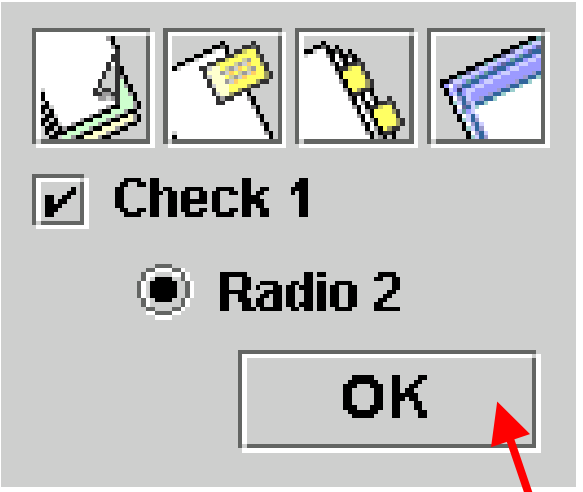


[4]

Button

- Einfachstes Widget
  - Wird benutzt, um eine bestimmte Aktion auszulösen
  - Z. B. um einen Befehl zu bestätigen ("OK") oder abzubrechen ("Abbruch")
- Klasse: **`JButton`**
- Erzeugen mit:

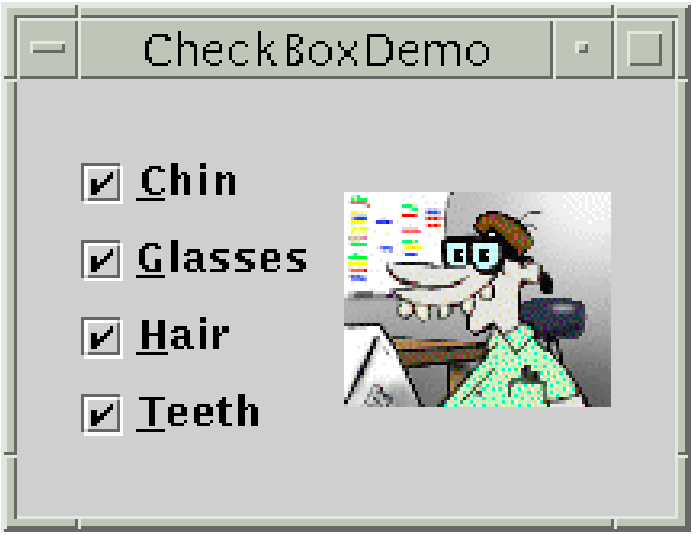
```
JButton b1 = new JButton("OK");
```



[4]

Check Box

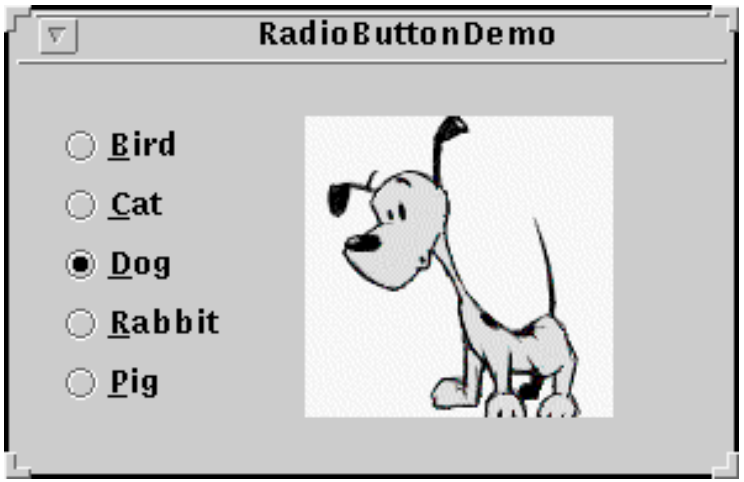
- Besteht aus einer kleinen, benannten Box zum Ankreuzen
  - Wird i. d. R. verwendet, um aus einer Palette von Optionen die gewünschten auszuwählen
  - Optionen sind häufig zu thematischen Gruppen zusammengefasst



[4]

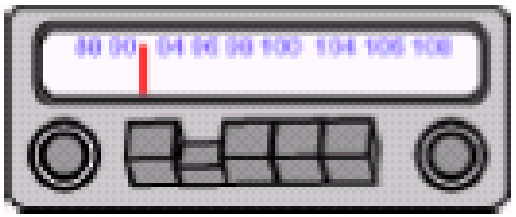
- Die gewählten Optionen sind durch einen Indikator, z. B. durch einen Haken oder ein kleines Kreuz, markiert
- Klasse: **`JCheckBox`**
- **`JCheckBox cb1 = new JCheckBox("Check Box");`**

Radio Button



[4]

- Besteht aus einem kleinen, benannten Kreis
- Konzept stammt vom Programm-wähler alter Kofferradios ab
  - Wird ein Button gedrückt, springt der gerade selektierte heraus
  - Radio Buttons sind nur als Gruppe sinnvoll
  - Nur eine Option in der Gruppe kann selektiert sein



  - Radio Buttons werden z. B. benutzt, um zwischen verschiedenen Werkzeugen oder Ausgabeformaten zu wählen
  - Klasse: **`JRadioButton`**
  - **`JRadioButton rb = new JRadioButton("Chan 1");`**

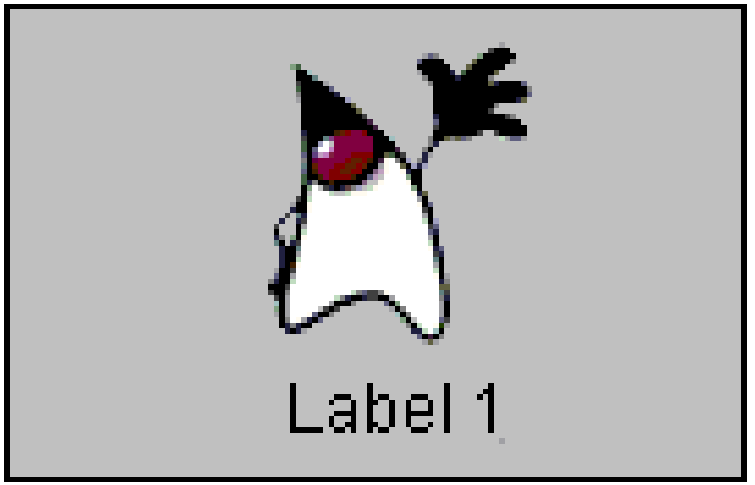
Radio Button Group

- Das Zuordnen einer Anzahl Radio Buttons zu einer Gruppe erfolgt durch ein **ButtonGroup**-Objekt
- Beispiel:

```
JRadioButton rb1 = new JRadioButton("Channel 1");
JRadioButton rb2 = new JRadioButton("Channel 2");
JRadioButton rb3 = new JRadioButton("Channel 3");
ButtonGroup group = new ButtonGroup();
group.add(rb1);
group.add(rb2);
group.add(rb3);
```

- Erst nach dem Zuordnen zu einer Gruppe funktioniert das Umschalten zwischen den Buttons

Label



[4]

- Mit Hilfe eines Labels können grafische Objekte benannt und ganz allgemein textuelle Informationen platziert werden
- Labels dienen z. B. dazu, Textfelder oder Gruppen von Check Boxes oder Radio Buttons zu benennen
  - Beispielsweise in einem Optionen-Menü
- Klasse: **`JLabel`**
- **`JLabel l = new JLabel("Ein Label");`**

Textfelder

- Textfelder erlauben die Eingabe einer einzelnen Textzeile oder ganzer Texte
  - Textzeile: **`JTextField`**
    - Beispiel:

```
JTextField tf = new JTextField("Start");
tf.setText("George Washington");
String text = tf.getText();
```

  - Textpassage: **`JTextArea(int rows, int columns)`**
    - Beispiel:

```
JTextArea ta = new JTextArea(5,20);
ta.setText("Thomas Jefferson");
String text = ta.getText();
```

[4]

Menü

- Zusammenfassung von Buttons in einem herunterklappbaren Bereich
- Erlaubt das Hinzufügen vieler Bedienelemente zu einer grafischen Benutzeroberfläche, ohne den Bildschirm zuzustellen

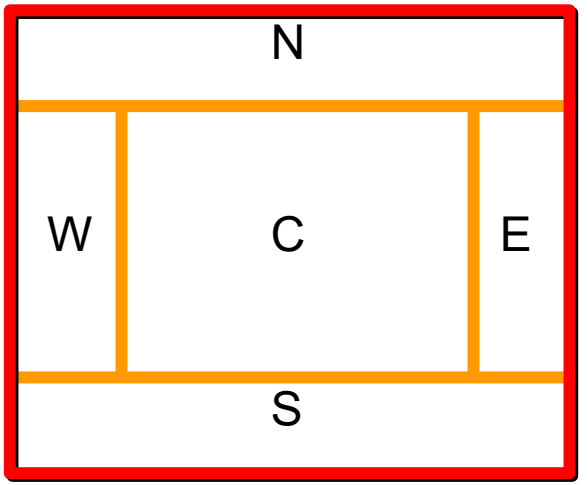
- Beim Design einer grafischen Benutzeroberfläche sollte immer darauf geachtet werden, dass die Arbeitsfläche nicht mit Informationen überfrachtet wird!

Widgets platzieren

- GUI-Programme basieren auf dem **`JFrame`**-Objekt
- Dieses repräsentiert einen Container für Widgets
- Die Platzierung der Widgets im Container erfolgt nach einem bestimmten Layout
- Das Standard-Layout heißt **`BorderLayout`**

java.awt.BorderLayout

- Widgets können in fünf verschiedenen Zellen platziert werden: Nord, Süd, Ost, West und Zentrum
  - **BorderLayout.NORTH**
  - **BorderLayout.SOUTH**
  - **BorderLayout.EAST**
  - **BorderLayout.WEST**
  - **BorderLayout.CENTER**
- Standardposition: Zentrum



Widgets zuweisen

- Widgets werden einem Container mit Hilfe der **`add(...)`**-Methode zugewiesen:

  ```
  add(Component comp, Object constraints)
  ```

  - Der Parameter **`constraints`** steuert die Position im Fenster
    - Mögliche Positionen beim **`BorderLayout`**:
      `NORTH, EAST, SOUTH, WEST, CENTER`
    - Bsp.: **`add(myButton, BorderLayout.CENTER);`**

- Mit **`remove(Component)`** können Widgets auch wieder aus einem Container entfernt werden

Beispiel: GUI bestehend aus einem Button

- Inhalt der Sketch-Datei des Programms

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

class DMProgGUI extends JFrame {

  private JButton b;

  DMProgGUI() {
    setTitle("Programm mit GUI");
    setSize(300, 300); // width, height; Größe GUI-Fenster
    b = new JButton("Button");
    add(b, BorderLayout.CENTER);
    setVisible(true);
  }
}

size(200, 200); // Größe des Processing-Grafikfensters
DMProgGUI app = new DMProgGUI();
```

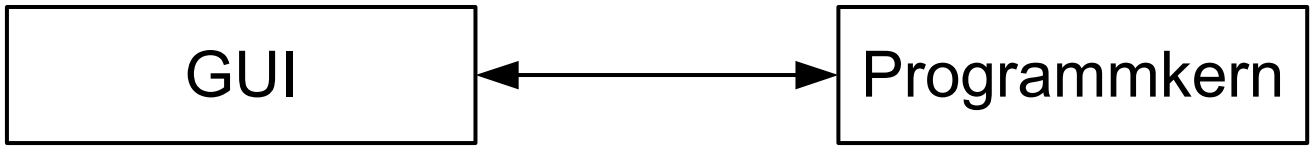
Ereignisbasierte Programmierung

Grafische Komponenten

- Grafische Komponenten sind Klassen, von denen Instanzen erzeugt werden
- Diese Instanzen werden visuell auf dem Bildschirm dargestellt und können mit der/dem Benutzer/in interagieren
- Die Interaktion funktioniert mit Hilfe der Computermaus oder der Tastatur auf der Grundlage von **Ereignissen**
- In diesem Zusammenhang wird von **ereignisbasierter Programmierung** gesprochen

Kommunizierende Komponenten

- Anwendung mit grafischer Bedienoberfläche:



Kommunikation

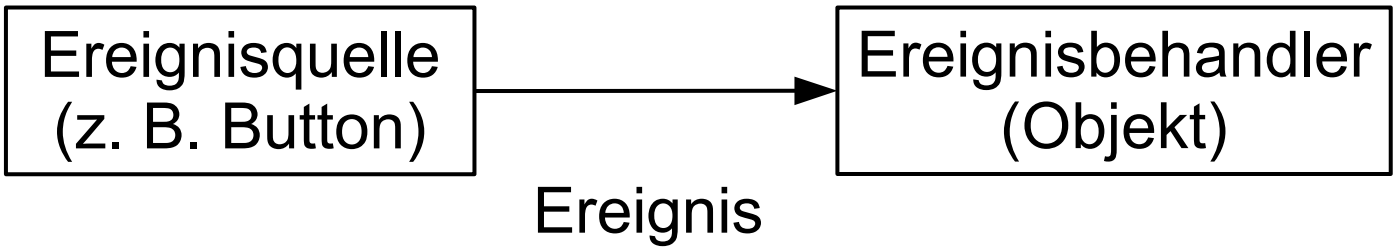
- Damit die Bedienoberfläche Aktionen auslösen kann, ist eine Kommunikationsverbindung zum Programmkern erforderlich
- Die Kommunikationsverbindung erlaubt dann einen Nachrichtenaustausch (engl.: *message passing*) zwischen der GUI und dem Programmkern

Kommunizierende Komponenten

- Der Nachrichtenaustausch wird mit Hilfe von Ereignissen durchgeführt:
  - Die GUI reagiert auf eine Benutzeraktion mit dem Auslösen eines Ereignisses
  - Der Programmkern reagiert dann in einer vorher festgelegten Weise auf dieses Ereignis
- Zu diesem Zweck existiert ein Ereignisbehandlungssystem (engl.: *event handling system*)

Ereignisbehandlung

- Ein Ereignis wird durch einen Mausklick, einen Tastendruck oder das Drücken eines Buttons ausgelöst
  - Ereignisquelle (engl.: *event source*)
- Im Programmkern führt dies zum Aufruf einer vordefinierten Methode in einem Ereignisbehandlungsobjekt
  - Ereignisbehandler (engl.: *event handler*), auch: Ereignishörer (engl.: *event listener*)



Ereignisquelle

- Eine Ereignisquelle (engl.: *event source*) ist ein Objekt, welches Ereignisse auslösen kann
- Ereignisquellen generieren spezifische Arten von Ereignissen
- Ereignisse repräsentieren in Java Instanzen zugehörender Klassen
- Ein Button generiert z. B. Objekte der Klasse **`ActionEvent`**
- Die Computermaus erzeugt **`MouseEvent`**-Objekte

Ereignisklassen

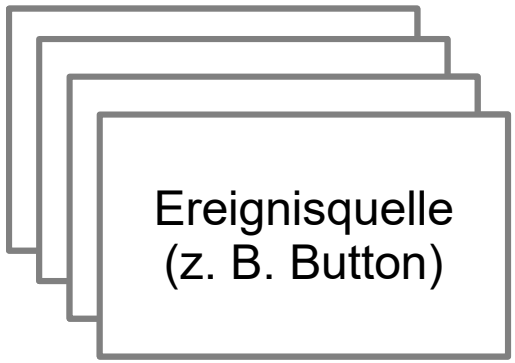
- **`ActionEvent, AdjustmentEvent, ComponentEvent, ContainerEvent, FocusEvent, HierarchyEvent, InputEvent, InputMethodEvent, InvocationEvent, ItemEvent, KeyEvent, MouseEvent, PaintEvent, TextEvent, WindowEvent`**
- Dokumentiert in der Java 2 Platform API Specification

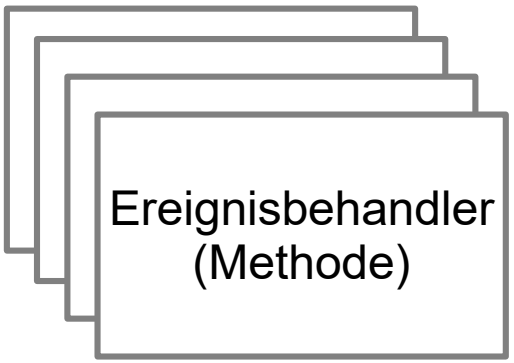
Ereignisbehandlung

- Erfolgt durch einen Ereignisbehandler (engl.: *event handler*), auch: Ereignishörer (engl.: *event listener*)
- Ein Ereignisbehandler ist ein Objekt, welches geeignet ist, Ereignisse eines bestimmten Typs entgegenzunehmen und zu behandeln
- Hierzu implementiert der Ereignisbehandler die Schnittstelle (engl.: *event handler / listener interface*) dieses Ereignistyps

Kommunikationsteilnehmer

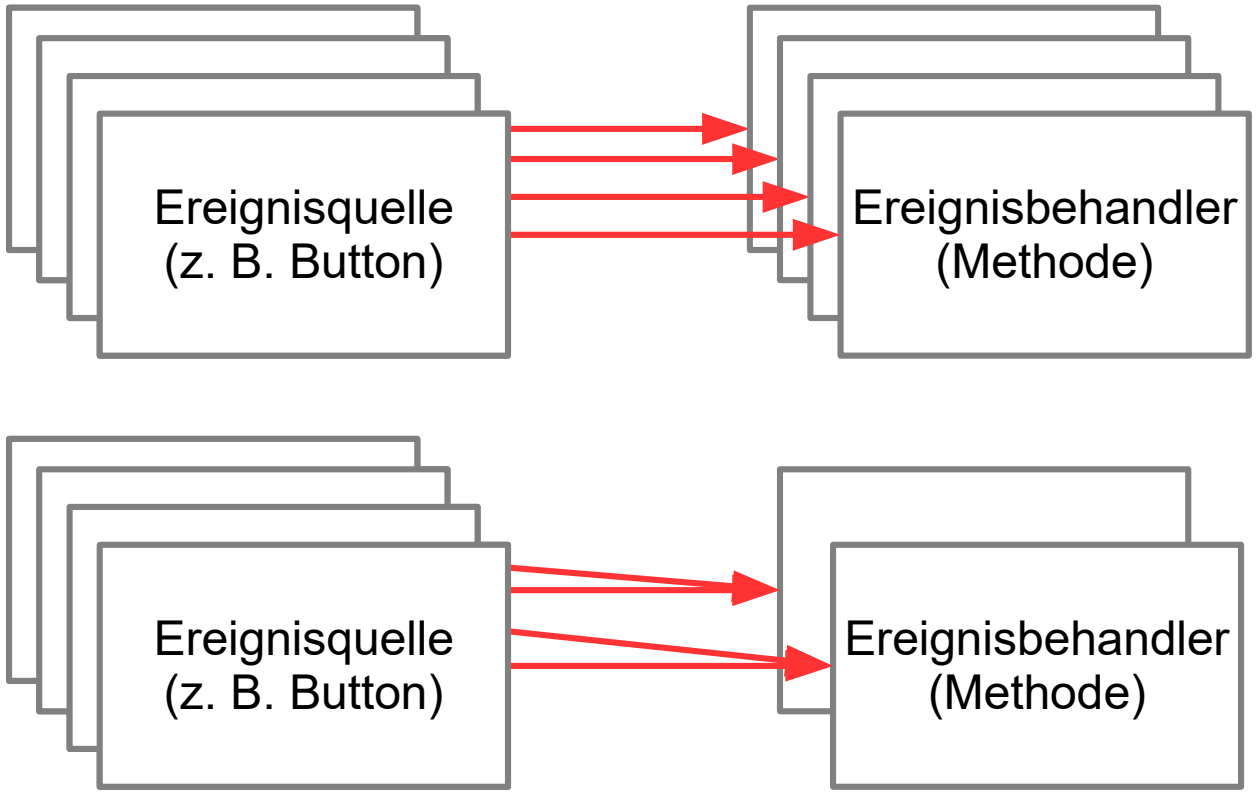
- Ein Programm besteht zunächst aus einer Anzahl nicht verbundener Ereignisquellen und -behandler





Kommunikationsverbindungen

- Um die Kommunikation zwischen den Ereignisquellen und -behandlern zu ermöglichen, müssen diese miteinander verbunden werden



Verbindung herstellen

- Zu diesem Zweck wird jeder Ereignisbehandler als Behandler für eine bestimmte Ereignisquelle registriert
- Hierfür besitzt die Ereignisquelle eine **addXXXListener()**-Methode
- Die Methode erhält eine Referenz auf den Behandler als Eingabeparameter
- Beispiel Button:

```
JButton b1 = new JButton("Button 1");
b1.addActionListener(new B1());
```

Verbindung herstellen

- In **`addXXXListener()`** repräsentiert das **`XXX`** den Typ des zu verarbeitenden Ereignisses
- Beispiel:
  - **`addActionListener()`** verbindet einen Behandler, der **`ActionEvents`** behandeln kann, mit einer Quelle für **`ActionEvents`**
  - **`ActionEvents`** werden u. a. von Buttons ausgelöst

Verbindung aufheben

- Es ist ebenfalls möglich, die Verbindung zwischen Ereignisquelle und -behandler während der Laufzeit wieder aufzuheben
  - Z. B. um die Ereignisquelle mit einem anderen Behandler zu verbinden
- Hierfür besitzen Ereignisquellen die `removeXXXListener()`-Methode
  - Z. B. `removeActionListener()`

Ereignisbehandlung

- Beispiel Button:

```
JButton b1 = new JButton("Button 1");
b1.addActionListener(new B1());
```

- Die Klasse **`B1`** implementiert eine für Ereignisse der Klasse **`ActionEvent`** vordefinierte Schnittstelle
- Teil dieser Schnittstelle ist eine so genannte Ereignisbehandlungsmethode

Schnittstellen

- Zu jedem Ereignistyp existiert eine zugehörende Schnittstelle
- Jeder Behandler von Ereignissen dieses Typs muss diese Schnittstelle implementieren
- Beispiel:
  - Die **`ActionListener`**-Schnittstelle gehört zu **`ActionEvents`**
- Allgemein bezeichnet **`XXXListener`** die Schnittstelle, die für die Behandlung von **`XXXEvents`** implementiert werden muss

Ereignisbehandlungsmethoden

- Jede dieser Schnittstellen definiert mindestens eine Ereignisbehandlungsmethode, die implementiert werden muss
- Ereignisbehandlungsmethoden werden aufgerufen, sobald das zugehörende Ereignis eingetreten ist
- Der Methodenrumpf beinhaltet die Anweisungen, die für die Behandlung des Ereignisses erforderlich sind
- Beispiel: Die `ActionListener`-Schnittstelle
  - Hier muss eine einzelne Methode implementiert werden: `actionPerformed(ActionEvent ae)`

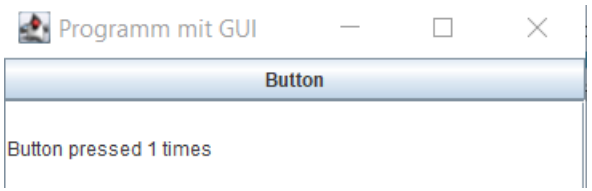
Beispiel: Button plus Textfeld (1)

```
// Ereignisbehandlung importieren
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class DMProgGUI extends JFrame {

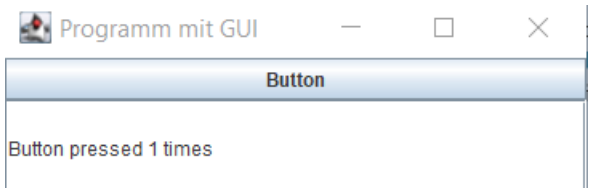
  // ...

}
```



Beispiel: Button plus Textfeld (2)

```
class DMProgGUI extends JFrame {
  private JButton b;
  private JTextField t;
  int i = 0;
  DMProgGUI() {
    setTitle("Programm mit GUI");
    setSize(300, 300); // width, height
    b = new JButton("Button");
    t = new JTextField(50);
    add(b, BorderLayout.NORTH);
    add(t, BorderLayout.CENTER);
    setVisible(true);
    b.addActionListener(new AL());
  }
  class AL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button pressed " + ++i + " times");
    }
  }
}
```



Innere Klasse

Ereignisbehandlung

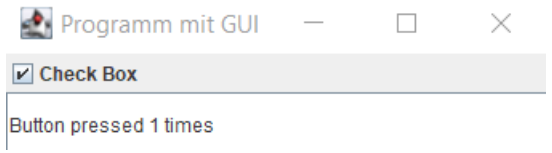
- Beispiel Button:

```
JButton b1 = new JButton("Button 1");
b1.addActionListener(new B1());
```

- Ein Objekt der Klasse **`B1`** wird hier dem Button **`b1`** als Ereignisbehandler zugewiesen
- Die Klasse **`B1`** implementiert eine für Ereignisse der Klasse **`ActionEvent`** vordefinierte Methode
  - **`actionPerformed(ActionEvent ae)`**
- "Ereignisbehandlungsmethode"

Beispiel: Check Box plus Textfeld

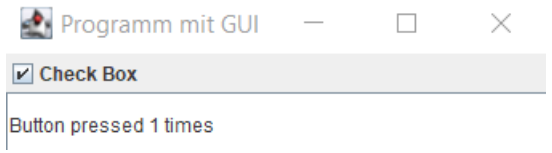
```
class DMProgGUI extends JFrame {
  private JCheckBox cb;
  private JTextField t;
  int i = 0;
  DMProgGUI() {
    setTitle("Programm mit GUI");
    setSize(300, 300); // width, height
    cb = new JCheckBox("Check Box");
    t = new JTextField(50);
    add(cb, BorderLayout.NORTH);
    add(t, BorderLayout.CENTER);
    setVisible(true);
    cb.addActionListener(new AL());
  }
  class AL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button pressed " + ++i + " times");
    }
  }
}
```



Beispiel: Anonyme innere Klasse

```
class DMProgGUI extends JFrame {
  private JCheckBox cb;
  private JTextField t;
  int i = 0;
  DMProgGUI() {
    setTitle("Programm mit GUI");
    setSize(300, 300); // width, height
    cb = new JCheckBox("Check Box");
    t = new JTextField(50);
    add(cb, BorderLayout.NORTH);
    add(t, BorderLayout.CENTER);
    setVisible(true);

    cb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          t.setText("Button pressed " + ++i + " times");
        }
      }
    );
  }
}
```



Das Ereignisobjekt

- Ereignisse führen bei ihrem Auftreten zur Erzeugung eines Ereignisobjekts
  - Dieses trägt spezifische Informationen über das Ereignis und seine Ursache in Form interner Daten
- Das Ereignisobjekt wird der Ereignisbehandlungsmethode als Eingabeparameter übergeben und kann im Methodenrumpf ausgewertet werden
- Beispiel: `actionPerformed(ActionEvent ae)`
  - Hier beinhaltet das Objekt `ae` alle relevanten Informationen über den aufgetretenen `ActionEvent`

Das Ereignisobjekt: Gelieferte Informationen

- Die Quelle des Ereignisses
  - Beispiele:
    - Eine Referenz auf den Button, den Radio Button oder die Check Box, die mit der Maus gedrückt wurde
    - Eine Referenz auf das Textfeld, in welches getippt wurde
  - Referenz auf die Quelle kann mit Hilfe der Methode **`getSource()`** ausgelesen werden
  - Beispiel:

```
if (ae.getSource() == b1) {
  /* do something */
}
```

Das Ereignisobjekt: Gelieferte Informationen

- Der Zeitpunkt, zu dem das Ereignis eingetreten ist
- Die (x,y)-Position, an der mit der Maus geklickt wurde
  - Relativ zum Ursprung des Koordinatensystems der betreffenden Komponente
- Die Taste, die gedrückt wurde (bei **`KeyEvents`**)
- Der Zustand der Sondertasten bei Tastendruck
  - Z. B. der Zustand der Umschalt- und Steuerungstaste

Ereigniskategorien

Ereigniskategorien

- Alle Ereignisse gehören zu zwei unterschiedlichen Kategorien:
- Semantische (auch: höhere) Ereignisse
  - (engl.: *semantic events*)
  - Repräsentieren Aktionen an denen GUI-Komponenten, wie z. B. Buttons, beteiligt sind
- Elementare Ereignisse
  - (engl.: *low-level events*)
  - Repräsentieren elementare Aktionen wie Mausklicks, Tastatureingaben oder das Öffnen, Verkleinern etc. von Fenstern

Semantische Ereignisse

Semantische Ereignisse

- Zu den semantischen Ereignissen gehören:
  - `ActionEvent`
  - `ItemEvent`
  - `TextEvent`
  - `AdjustmentEvent`
- **Hauptmerkmal**: Die Schnittstelle für die Ereignisbehandlung umfasst nur eine **einzelne** Methode

`ActionEvent`

- Wird z. B. durch Drücken eines Buttons, Auswählen eines Menüpunktes oder durch Eingabe von Text in ein Textfeld ausgelöst

- Ereignis wird durch die **`actionPerformed()`** Methode der **`ActionListener`**-Schnittstelle behandelt

  - Ereignisobjekt erlaubt es, die Quelle des Ereignisses, z. B. den Button oder Menüpunkt, zu identifizieren (**`getSource()`**)

`ItemEvent`

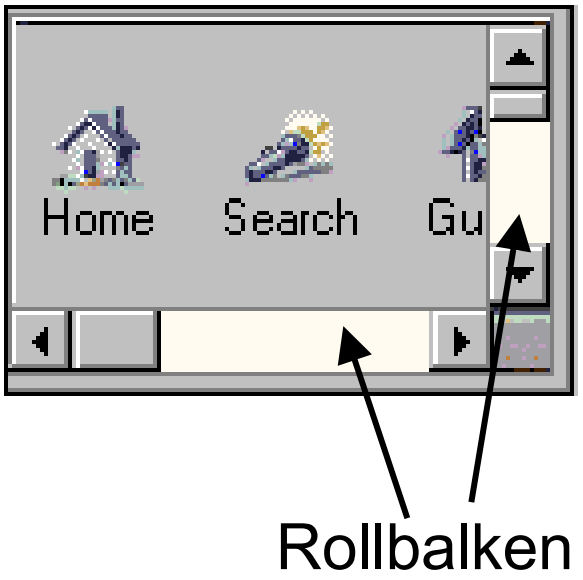
- Wird z. B. durch den Zustandswechsel einer Check Box (**`JCheckBox`**) ausgelöst
  - Check Boxes generieren bei Zustandswechsel einen Action Event sowie einen Item Event
- Wird durch die **`itemStateChanged()`** Methode der **`ItemListener`**-Schnittstelle behandelt
- Neben der Identifikation der Ereignisquelle (**`getItemSelectable()`**) erlaubt das Ereignisobjekt die Bestimmung des neuen Zustandes der Check Box (**`getStateChange()`**)

TextEvent

- Wird generiert, wenn sich der Text einer editierbaren Textkomponente geändert hat
- Alternative zur direkten Auswertung von Tastatureingaben
- Wird von der `textValueChanged()` Methode der `TextListener`-Schnittstelle behandelt

AdjustmentEvent

- Wird ausgelöst durch Bewegen von Rollbalken
- Liefert Informationen über die Art der Änderung (**`getAdjustmentType()`**):
  - UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK
- Wird an **`adjustmentValueChanged()`** Methode der **`AdjustmentListener`**-Schnittstelle übergeben



Elementare Ereignisse

Elementare Ereignisse

- Zu den elementaren Ereignissen gehören bei Java:
  - `MouseEvent`
  - `ComponentEvent`
  - `FocusEvent`
  - `ContainerEvent`
  - `PaintEvent`
  - `WindowEvent`
- **Hauptmerkmal**: Schnittstelle für die Ereignisbehandlung umfasst unter Java **mehrere** Methoden!
- Wir betrachten die Behandlung elementarer Ereignisse hier nur für Processing

Animation unter Processing

- Geschieht ebenfalls über Ereignisbehandlung
- Hier über die Auswertung des Bildwechselereignis
- Die zugehörende Ereignisbehandlungsmethode heißt **`draw()`**
  - **`draw()`** wird automatisch von Processing zyklisch aufgerufen
  - Sobald die Methode **`draw()`** in einer Sketch-Datei vorhanden ist, wird auch die Konstruktor-Methode **`setup()`** benötigt
  - In der **`setup()`**-Methode steht dann das Hauptprogramm

Beispiel aus Processing-Dokumentation

- Siehe https://processing.org/reference/draw_.html:

```
float yPos = 0.0;

void setup() {    // setup() wird einmal aufgerufen
  size(200, 200);
  frameRate(30); // Festlegung der Bildrate in fps
}

void draw() {     // wird zyklisch aufgerufen
  background(204);
  yPos = yPos - 1.0;
  if (yPos < 0) { yPos = height; }
  line(0, yPos, width, yPos);
}
```

Mausereignisse unter Processing

- **`draw()`** ist auch die Grundlage für die Verarbeitung von Mausereignissen
-
```
// Although empty here, draw() is needed so
// the sketch can process user input events
// (mouse presses in this case).

void draw() { }

void mousePressed() {
  line(mouseX, 10, mouseX, 90);
}
```
- Weitere Mausereignisse:

  siehe
  `https://processing.org/reference#input-mouse`

Beispiel: Mausklick im Grafikfenster auswerten

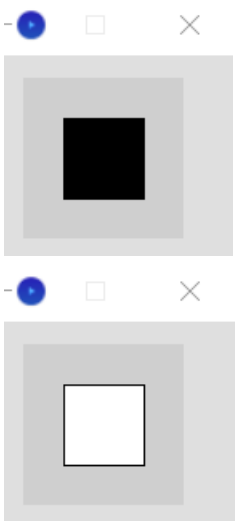
```
// Click within the image to change the value of
// the rectangle after the mouse has been clicked

int value = 0;

void draw() {
  fill(value); // Color used to fill subsequent shapes
  rect(25, 25, 50, 50);
}

// void mouseClicked() { Originalbeispiel

void mousePressed() {//reagiert evtl. besser
  if (value == 0)
    value = 255;
  else
    value = 0;
}
```



Tastaturereignisse auswerten

```
void draw() {} // Required for input event processing

void keyPressed() { // alternativ: void keyReleased()
  /* The system variable key always contains the value
  of the most recent key on the keyboard that was used */
  switch(key) {
    case 'a':
      println("a");
      break;
    case 'b':
      println("b");
      break;
    default:
      println("unknown");
      break;
  }
}
```

Container und Komponenten

Wie werden grafische Komponenten benutzt?

- Grundlage einer GUI ist zunächst eine Klasse, die **`JFrame`** erweitert
  - Repräsentiert den (übergeordneten) "Hauptcontainer"
- Danach müssen die erforderlichen grafischen Komponenten entsprechend des angestrebten Layouts der GUI ausgewählt werden
  - Buttons, Textfelder, etc.

javax.swing.**JComponent**

- Unter Java-Swing sind alle Komponenten Unterklassen der Klasse **`JComponent`**
- **`JComponent`** als Oberklasse wird erweitert durch die Klassen:
  - **`JButton`**, **`JLabel`**, **`JRadioButton`**, **`JCheckbox`**, **`JMenuBar`**, etc.
  - Die Unterklassen erben allgemein verwendbare Methoden von **`JComponent`**
    - Beispiel: **`setVisible(boolean aFlag)`**
      - Erlaubt es, die Komponente sichtbar oder unsichtbar zu schalten

`JComponent` – Graphik-Attribute

- Font
- Vordergrund- und Hintergrundfarbe
- Minimale, maximale und bevorzugte Größe
  - Momentane Größe abfragen und bevorzugte Größe einstellen:
  - **`getSize(Dimension size)`**, **`setPreferredSize(Dimension preferredSize)`**
  - Erfordert einen Layout Manager, der die bevorzugte Größe respektiert (z. B. **`FlowLayout`**)

`JComponent` – Graphik-Attribute

- Cursortyp (z. B. default, cross-hair, text, wait, move) abfragen und einstellen
  - **`getCursor()`**
  - **`setCursor(Cursor cursor)`**
- Zugriffs- und Änderungsmethoden sind für die meisten Attribute verfügbar
  - **`getXXX(...), setXXX(...)`**

`JComponent` - Enabled

- Erlaubt Kontrolle über das Verhalten der GUI zur Laufzeit des Programms
- GUI-Elemente können entsprechend des gegenwärtigen Programmzustandes aktiviert oder deaktiviert werden
  - **`setEnabled(boolean enabled)`**
- Nur aktivierte Komponenten können auf Benutzereingaben reagieren und das Programm steuern
- Beispiel: Deaktivieren der meisten Menübefehle, solange noch keine Datei zum Bearbeiten geöffnet wurde

`JComponent` - Visibility

- GUI-Elemente können ebenfalls in Abhängigkeit vom Programmstatus sichtbar oder unsichtbar geschaltet werden
  - `setVisible(boolean visible)`
- Beispiel: Anpassen der verfügbaren Menübefehle an erfahrene Nutzer oder Anfänger

Wie werden Komponenten benutzt?

- Bevor eine grafische Komponente benutzt werden kann, muss sie erzeugt werden
- Zu diesem Zweck instanziert man die zugehörende Komponentenklasse
- Beispiel: Erzeugen eines Standardbuttons

```
JButton b = new JButton("OK");
```

Beschriftung

- Danach wird die Komponente einem Container zugewiesen

Komponenten einem Container zuweisen

- Hierfür stellen die Containerklassen **`add(...)`**-Methoden zur Verfügung:

```
add(Component comp)
add(Component comp, Object constraints)
```

  - Der Parameter **`constraints`** steuert das Layout im Container

- Mit **`remove(Component)`** können Komponenten auch während der Laufzeit aus einem Container entfernt werden

Anordnung der Komponenten

- Die grundlegende Anordnung der Komponenten in der Benutzeroberfläche wird durch Wahl des oder der Layout Manager festgelegt
- Zusätzlich wird die Anordnung durch die Reihenfolge der **`add()`**-Aufrufe bestimmt
- Jedem Container ist ein Layout Manager zugeordnet
- Dieser kann frei gewählt werden
  - Wird kein Layout Manager gewählt, so wird der voreingestellte Manager benutzt
  - Bei Swing ist dies das **`BorderLayout`**

Layout Manager

Was ist ein Layout Manager?

- Layout Manager kontrollieren, wie eine grafische Komponente in einem Container platziert wird
- Sie ermöglichen ebenfalls eine automatische Anpassung der Anordnung der GUI-Elemente an die Fenstergröße der Anwendung
  - Eine Java-GUI kann z. B. auf einem Desktop-Rechner, Smartphone oder einem Tablet zur Anwendung kommen
  - Dann ist die Anordnung der GUI-Elemente an die Größe und das Seitenverhältnis des Anwendungs-Fensters anzupassen
  - Java-Anwendungen unterstützen Responsivität

Was ist ein Layout Manager?

- Java stellt eine Reihe unterschiedlicher Layout Manager bereit
- Die Art, wie grafische Komponenten in einem Container angeordnet werden, hängt vom gewählten Layout Manager ab
- Bei bestimmten Managern wird das Layout automatisch an die Größe des Programmfensters angepasst
  - Wird z. B. die Größe des Fensters im laufenden Betrieb geändert, so kann dies eine Neuanordnung der grafischen Komponenten nach sich ziehen

Wahl eines Layout Managers

- Der gewünschte Layout Manager wird entweder bei der Erzeugung eines Containers als Konstruktorparameter angegeben oder durch Aufruf der Methode **`setLayout()`** dem Container zugewiesen
- **`setLayout(mgr)`**
  - Container und Layout Manager zusammen implementieren eine bestimmte "Layout-Taktik" (engl.: *layout policy*)
- Beispiele:

```
JFrame aFrame = new JFrame("My Frame");
aFrame.setLayout(new GridLayout(3,3));
JPanel myPanel = new JPanel(new GridLayout(1,5));
```

java.awt.FlowLayout

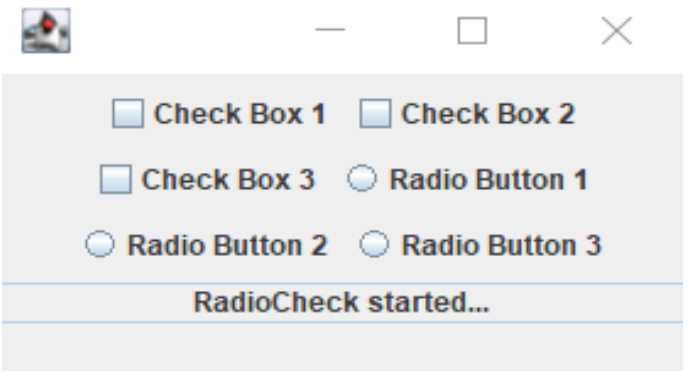
- Komponenten werden von links nach rechts und von oben nach unten im Container platziert
  - Voreingestellter Layout Manager des AWT
- Mögliche Ausrichtungen:
  - **`FlowLayout.CENTER`**: Komponenten zentriert (Voreinstellung)
  - **`FlowLayout.LEFT`**: Komponenten links ausgerichtet
  - **`FlowLayout.RIGHT`**: Komponenten rechts ausgerichtet
  - Jeweils pro Reihe

java.awt.**FlowLayout**

- Ausrichtung und Abstände (engl.: *Gap*) der Komponenten können als Konstruktorparameter oder mit Hilfe von Methoden der Klasse vorgegeben werden
  - **`FlowLayout(int align, int hgap, int vgap)`**
  - **`setAlignment(int align)`**
  - **`setHgap(int hgap)`**, **`setVgap(int vgap)`**
  - Abstände werden in Pixel angegeben

Beispiel: Flow Layout (1)

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




```
class RadioCheck extends JFrame {
  JTextField t = new JTextField(50);
  JCheckBox cb1 = new JCheckBox("Check Box 1");
  JCheckBox cb2 = new JCheckBox("Check Box 2");
  JCheckBox cb3 = new JCheckBox("Check Box 3");

  ButtonGroup g = new ButtonGroup();
  JRadioButton rb1 = new JRadioButton("RB 1", false);
  JRadioButton rb2 = new JRadioButton("RB 2", false);
  JRadioButton rb3 = new JRadioButton("RB 3", false);
```

Beispiel: Flow Layout (2)

```
RadioCheck() {
  setSize(300, 300); // Größe des GUI-Fensters
  t.setEditable(false);
  t.setHorizontalAlignment(JTextField.CENTER);
  t.setFont(new Font("Arial",Font.BOLD,12));
  t.setText("RadioCheck started...");
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  cp.add(cb1); cp.add(cb2); cp.add(cb3);
  g.add(rb1); g.add(rb2); g.add(rb3);
  cp.add(rb1); cp.add(rb2); cp.add(rb3);
  cp.add(t);
  setVisible(true); // GUI anzeigen
}}
void setup() {RadioCheck app = new RadioCheck();}
```

java.awt.GridLayout

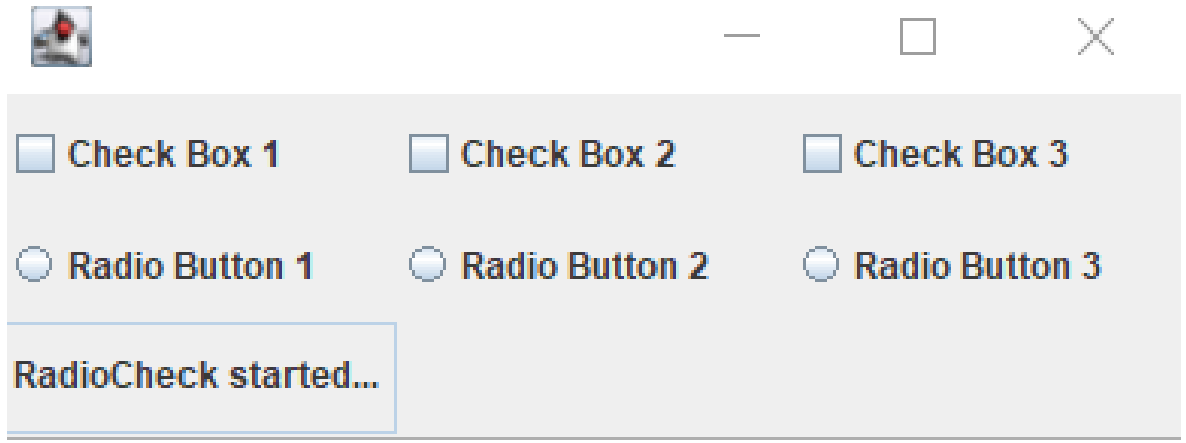
- Platziert Komponenten in einer Tabelle
  - Jede Zelle hat die gleiche Größe
  - Jede Komponente füllt den gesamten Platz in ihrer Zelle aus

java.awt.GridLayout

- Anzahl der gewünschten Zeilen und Spalten werden als Konstruktorparameter angegeben:
  - **`GridLayout(int rows, int cols)`**
  - **`GridLayout(int rows, int cols,`**
    `int hgap, int vgap)`
- Beispiel: Radio Buttons und Check Boxes mit Grid Layout

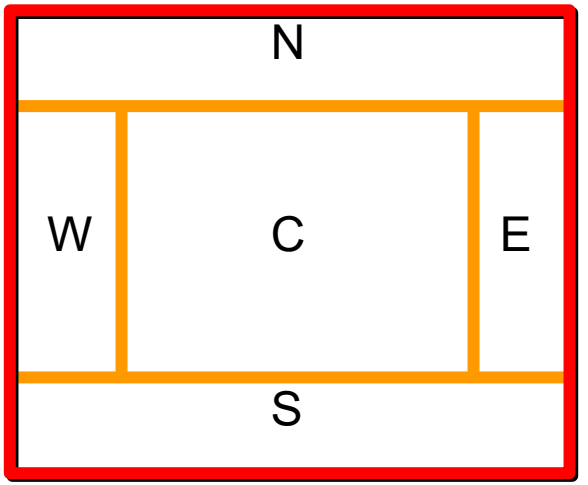
```
Container cp = getContentPane();
cp.setLayout(new GridLayout(3,3));
```

- Ansonsten wie zuvor



java.awt.BorderLayout

- Komponenten können in fünf verschiedenen Zellen platziert werden: Nord, Süd, Ost, West und Zentrum
  - **BorderLayout.NORTH**
  - **BorderLayout.SOUTH**
  - **BorderLayout.EAST**
  - **BorderLayout.WEST**
  - **BorderLayout.CENTER**



- Standard Layout Manager unter Swing
  - Standardposition: Zentrum

`BorderLayout` – Platzierung von Komponenten

- Um die Komponente wie gewünscht zu platzieren, ist es erforderlich, die Position beim Aufruf von **`add()`** anzugeben
  - Hierin unterscheidet sich der **`BorderLayout`** Manager von den bisherigen Managern

- Beispiel:

```
JTextField t = new JTextField(50);
cp.add(BorderLayout.NORTH,t);
```

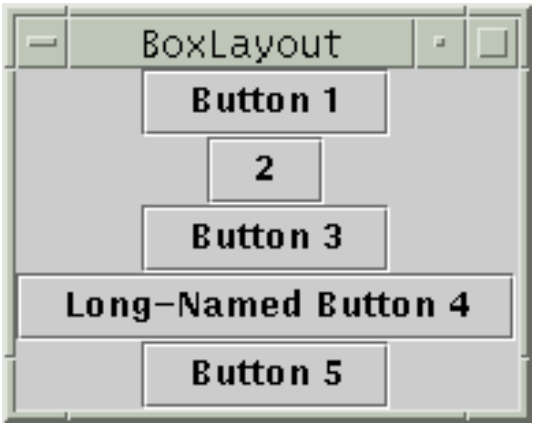
- Wird dies vergessen, so werden alle Komponenten übereinander im Center-Feld „gestapelt“

Radio Buttons und Check Boxes: Border Layout

```
Container cp = getContentPane();
// cp.setLayout(new BorderLayout());  nicht nötig
// ...
// Nicht genug Platz für alle Komponenten vorhanden
cp.add(BorderLayout.NORTH,t);
cp.add(BorderLayout.WEST,rb1);
cp.add(BorderLayout.CENTER,rb2);
cp.add(BorderLayout.EAST,rb3);
cp.add(BorderLayout.SOUTH,cb1);
// ...
```

Andere Layout Manager

- BoxLayout
  - Anordnung der Komponenten in einer einzigen Zeile oder Spalte



- CardLayout
  - Unterstützt eine zeitlich veränderbare Anordnung der Komponenten
  - Komponenten bilden einen Stapel von "Karten"
  - Sichtbar ist nur die Komponente der gerade aktiven, d. h. sichtbaren Karte

Andere Layout Manager

- GridBagLayout
  - Flexibelster Layout Manager
  - Erlaubt relativ freies Platzieren der Komponenten im Container
  - Primär intendiert für Benutzung mit GUI-Builder

- Eigene Layout Manager
  - Es ist auch möglich, durch Implementieren des Interfaces **`LayoutManager`** eigene Layout Manager zu konstruieren

Layouts kombinieren

- In der gleichen Benutzeroberfläche können unterschiedliche Layouts miteinander kombiniert werden
- Dies geschieht über die Erstellung einer entsprechenden Anzahl von Panels jeweils mit eigenem Layout Manager
- Auf diese Weise lassen sich beliebig viele Komponenten unterbringen

BorderLayout mit GridLayout kombiniert

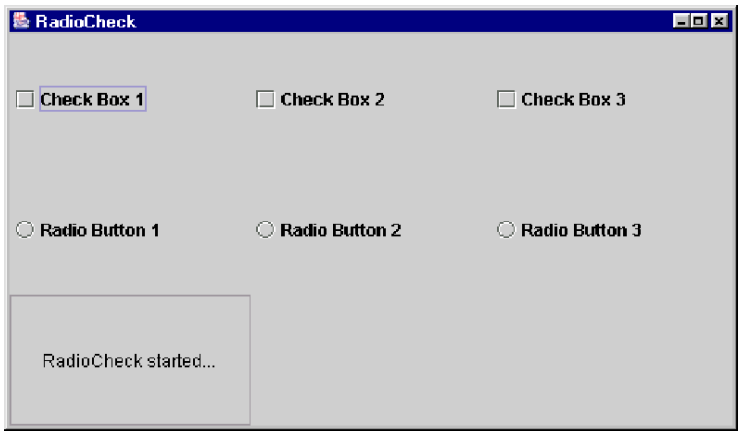
```
class PanelApp extends JFrame {
  JCheckBox cb1 = new JCheckBox("Check Box 1");
  JCheckBox cb2 = new JCheckBox("Check Box 2");
  JCheckBox cb3 = new JCheckBox("Check Box 3");

  PanelApp() {
    setSize(300, 300);
    Container cp = getContentPane();
    JPanel p = new JPanel(new GridLayout(1, 3));
    p.add(cb1);
    p.add(cb2);
    p.add(cb3);
    cp.add(p, BorderLayout.CENTER);
    setVisible(true); // GUI anzeigen
  }
}
```

Ikonen

Ikone

- Bis jetzt besaßen die in den Beispielen verwendeten GUI-Komponenten nur ihr voreingestelltes Aussehen
- Die Buttons waren z. B. nur mit einem Namen versehen



- Buttons lassen sich ebenfalls mit Ikonen (engl: *Icons*) ausstatten
  - Dies ermöglicht eine visuell ansprechendere, ergonomische und auch kompaktere Darstellung
    - Insbesondere, wenn eine Vielzahl von Komponenten in einem Container platziert werden sollen

Ikonen konstruieren: Applikation

- Ikonen werden mit Hilfe z. B. eines GIF- oder PNG-Bildes konstruiert, dessen Name einer Instanz der **`ImageIcon`**-Klasse übergeben wird

- Beispiel:

```
Icon myIcon = new ImageIcon(String filename)
```

  - Dies generiert eine Ikone **`myIcon`** aus einer Bilddatei mit dem Namen **`filename`**
  - **`filename`** kann zusätzlich eine Pfadangabe enthalten

Ikonen an Komponente binden

- Nachdem eine Ikone geladen wurde, kann diese an einen Button gebunden werden
- Z. B. beim Erzeugen des Buttonobjekts
- Zu diesem Zweck existieren eine Reihe unterschiedlicher Konstruktoren

Radio Button: Beispiele für Konstruktoren

- **`JRadioButton(Icon icon)`**
  - Erzeugt einen zu Beginn nicht selektierten Radio Button ohne textuelle Beschreibung mit der angegebenen Ikone
- **`JRadioButton(Icon icon, boolean selected)`**
  - Erzeugt einen Radio Button ohne textuelle Beschreibung mit der angegebenen Ikone im gewählten Anfangszustand
- **`JRadioButton(String text, Icon icon, boolean selected)`**
  - Erzeugt einen Radio Button mit textueller Beschreibung und der angegebenen Ikone im gewählten Anfangszustand

Radio Button Ikone zuweisen

- Anstatt zu schreiben:

```
JRadioButton rb = new JRadioButton("B1", false);
```

- wird jetzt dem Konstruktor eine Referenz auf ein **ImageIcon**-Objekt übergeben:

```
JRadioButton rb  = new JRadioButton(myIcon, false);
```

- Zusätzlich kann eine Beschriftung festgelegt werden:

```
JRadioButton rb  = new JRadioButton(
  "B1", myIcon, false
)
```

Beispiel: Drei Radio Buttons mit Ikonen

```
public class RadioIcon extends JFrame {
  // Radio Buttons etc. wie zuvor
  Icon [] icon = new Icon[3];
  public RadioIcon() {
    setSize(300, 300);
    //...
    String str = dataPath(""); // data-Pfad von Process.
    icon[0] = new ImageIcon(str + "/icons/icon1.png");
    icon[1] = new ImageIcon(str + "/icons/icon2.png");
    icon[2] = new ImageIcon(str + "/icons/icon3.png");
    for (int i=0; i<3; i++) {
      rb[i] = new JRadioButton(icon[i],false);
      //...
    }
}
```

`data`-Verzeichnis

- Mediendateien werden unter Processing in das **`data`**-Verzeichnis gelegt
  - Unterverzeichnis des Verzeichnisses, in dem die Sketch-Datei des Programms liegt
- Die Zugriffsmethoden von Processing setzen dieses Verzeichnis voraus

Buttons: Zustände

- Buttons können sich in unterschiedlichen Zuständen befinden:
  - Unter der Maus (engl.: *Roll Over*)
  - Gedrückt
  - Selektiert und nicht selektiert
  - Aktiviert und deaktiviert
    - Jeweils im selektierten und nicht selektierten Zustand
- Für jeden dieser Zustände kann dem Button eine gesonderte Ikone zugewiesen werden

Zustandsabhängige Ikonen

- Die folgenden Methoden sind für die Klasse **`AbstractButton`** definiert und somit für alle Arten von Buttons gültig
  - **`AbstractButton`** ist Oberklasse aller Buttons

- **`setIcon(Icon defaultIcon)`**
  - Definiert die Standardikone des Buttons
  - Diese wird üblicherweise beim Instanzieren des Buttons bereits zugewiesen, sodass **`setIcon`** nicht erforderlich ist

Zustandsabhängige Ikonen

- Die folgenden zustandsabhängigen Ikonen werden nach der Instanzierung des Buttons durch die jeweils angegebene Methode zugewiesen:
- **`setPressedIcon(Icon pressedIcon)`**
  - Definiert die Ikone für den gedrückten Zustand
- **`setSelectedIcon(Icon selectedIcon)`**
  - Definiert die Ikone für den selektierten Zustand
- **`setDisabledIcon(Icon disabledIcon)`**
  - Definiert die Ikone für den deaktivierten Zustand

Zustandsabhängige Ikonen

- **`setRolloverIcon(Icon rolloverIcon)`**
  - Definiert die Ikone für den Roll-Over-Effekt
- **`setRolloverEnabled(boolean b)`**
  - Schaltet den Roll-Over-Effekt ein (**`true`**) oder aus (**`false`**)
- **`setDisabledSelectedIcon(Icon disSelIcon)`**
  - Definiert die Ikone für den selektierten und deaktivierten Zustand
- **`setRolloverSelectedIcon(Icon rollOverSelIcon)`**
  - Definiert die Ikone für den Roll-Over-Effekt beim selektierten Button

Beispiel: Radio Button

```
// Assign the icon of the pressed state
rb.setPressedIcon(pressedIcon);

// Assign the icon of the selected state
rb.setSelectedIcon(selectedIcon);

// Assign the icon of the disabled state
rb.setDisabledIcon(disabledIcon);

// Assign the icon of the disabled selected state
rb.setDisabledSelectedIcon(disabledSelectedIcon);

// Enable roll over
rb.setRolloverEnabled(true);

// Assign the icon of the roll over state
rb.setRolloverIcon(rolloverIcon);
```

Ton

Ton

- Toninhalte können sowohl über Java oder Processing abgespielt werden
- Möglich mit Hilfe der "Java Sound Engine"
- Die Java Sound Engine besteht tatsächlich aus zwei verschiedenen Maschinen:
  - Die Audio Rendering Engine
    - Unterstützt WAV (PC), AIFF (Mac), AU (Unix)
  - Die MIDI Sound Synthesis Engine
    - Unterstützt MIDI Type 0, MIDI Type 1, Rich Music Format (RMF)
- Wir betrachten hier den Ablauf unter Processing

Processing

- Aufnehmen, abspielen, analysieren und synthetisieren von Audioinhalten
- Umgesetzt mit der Sound Library und Minim Library
- Verfügbar ist u. a.:
  - Eine Sammlung von Oszillatoren für grundlegende Wellenformen,
  - verschiedene Rauschgeneratoren sowie
  - Effekte und Filter zur Veränderung von Audiodateien und anderen generierten Sounds

Processing: Audiodateien abspielen

- Bevor eine Audiodatei abgespielt werden kann, muss diese zunächst geladen werden
- Unterstützte Dateiformate: WAV, AIF/AIFF und MP3
  - WAV or AIF sind weniger aufwendig zu decodieren als MP3
    - Ggf. beachten bei einfacher Android-Hardware
  - WAV-Dateien werden am zuverlässigsten abgespielt

Audiodateien unter Processing abspielen (1)

```
import processing.sound.*;
SoundFile file;

void setup() {
  size(640, 360);
  background(255);

  // Load a soundfile from the /data folder
  // of the sketch and play it back
  // Data folder is home of media files
  file = new SoundFile(this, "sample.mp3");
  file.play();
}

void draw() {}
```

- Audiodateien werden als `SoundFile`-Objekte geladen

Audiodateien unter Processing abspielen (2)

- Das Abspielen der Audiodatei wird durch Aufruf der `play()` Methode eingeleitet
- Weitere Methoden:
  - `loop()`
    - Spielt Audiodatei wiederholt ab
  - `pause()`
    - Unterbricht das Abspielen der Audiodatei
  - `cue(float time)`
    - Springt zu der angegebenen Position (in sec.) in der Datei
- Weitere Informationen siehe https://processing.org/reference/libraries/sound/SoundFile.html

Audiodateien unter Processing abspielen (3)

- Mit der SoundFile-Klasse geladene Audiodateien werden vollständig in den Speicher geladen
- Ein Sketch benötigt ca. 20 MB RAM pro Minute Stereo-Audio
- Speicher mindestens doppelt so groß wählen wie das größte im Programm verwendete Audio-Sample
  - Ggf. Memory vergrößern in **`preferences.txt`** mit „Datei“ > „Einstellungen“ falls **`OutOfMemoryError`**
- Abhängig vom Format und der Länge der Audiodateien kann das erstmalige Laden den Sketch für mehrere Sekunden blockieren. SoundFile-Objekte daher in **`setup()`** erstellen
- Dekodierung komprimierter Formate (mp3, ogg usw.) kann auf Raspberry Pi recht langsam sein (20s für eine 3min. MP3-Datei, 14 Sekunden für ogg auf einem Raspberry Pi 3B 32bit).
  - Generell wird WAV-Format empfohlen, da Audio-Samples ohnehin unkomprimiert im RAM gespeichert werden

Tonaufnahmen

- Möglich mittels der Minim-Bibliothek:
  `AudioInput`-Objekt definiert Tonquelle
- Konstruktoren:
  - **`AudioInput getLineIn()`**
    - Default: Stereo, 1024 sample buffer, 44100kHz, 16bit
    - Normalerweise ausreichend
    - Type: Minim.MONO or Minim.STEREO
  - **`AudioInput getLineIn(int type)`**
  - **`AudioInput getLineIn(int type, int bufferSize)`**
  - **`AudioInput getLineIn(int type, int bufferSize, float sampleRate)`**
  - **`AudioInput getLineIn(int type, int bufferSize, float sampleRate, int bitDepth)`**

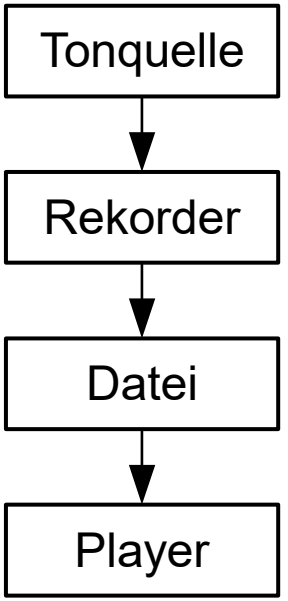
Rekorder

- Methode **`createRecorder`** definiert Rekorder für die Audioquelle und Zieldatei
- Beispiel:

```
void setup()
{
  minim = new Minim(this);
  in = minim.getLineIn(Minim.MONO); // Mono-Mikrofon
  // Zieldatei angeben
  recorder = minim.createRecorder(in,
                  "data/afile.wav", true);
}
```

Ton aufzeichnen, speichern und abspielen

- Am einfachsten durch Auswertung von Tastaturereignissen mit dem folgenden Grundgerüst:




```
void keyReleased() {
  switch(key) {
    case 'r':
      if ( recorder.isRecording() )
        recorder.endRecord();
      else
        recorder.beginRecord();
    break;
    case 's':
      recorder.save();
      break;
    case 'p': // lädt afile.wav aus data
       AudioPlayer player = minim.loadFile("afile.wav");
       player.play();
}} // Recorder ist kein Player, darum erst speichern
```

Toninhalte visualisieren: Wellenform (1)

- Beispiel aus der Minim-Dokumentation: Ton während der Aufzeichnung als Welle visualisieren
- Setup:

```
import ddf.minim.*;
Minim minim;
AudioInput in;
void setup() {
  size(512, 200);
  minim = new Minim(this);

  // use getLineIn to get an AudioInput
  in = minim.getLineIn();
}
```

Toninhalte visualisieren: Wellenform (2)

- Darstellung mit draw:

```
void draw() {
  background(0); // schwarzer Hintergrund
  stroke(255);   // weiße Wellenlinien
  // draw the waveforms of what we are monitoring
  for(int i = 0; i < in.bufferSize() - 1; i++) {
    line( i, 50 + in.left.get(i)*50,
          i+1, 50 + in.left.get(i+1)*50);
    line( i, 150 + in.right.get(i)*50,
          i+1, 150 + in.right.get(i+1)*50 );
  }
  String monitoringState = in.isMonitoring() ?
                           "enabled" : "disabled";
  text("Input monitoring is currently " +
       monitoringState + ".", 5, 15 );
}
```

Toninhalte visualisieren: Wellenform (3)

- Monitoring, d. h. Wiedergabe des aufgenommenen Tons, während der Visualisierung ein- und ausschalten

```
void keyPressed() {
  if ( key == 'm' || key == 'M' ) {
    if ( in.isMonitoring() )
      in.disableMonitoring();
    else
      in.enableMonitoring();
  }
}
```

Toninhalte visualisieren: Wellenform (4)

- Tonaufzeichnung aus Datei während des Abspielens als Welle visualisieren
- Setup:

```
import ddf.minim.*;
Minim minim;
AudioPlayer in;
void setup() {
  size(512, 200);
  minim = new Minim(this);
  in = minim.loadFile("audiofile.wav");
  in.play();
  println("Playing.");
}
```

Toninhalte visualisieren: Wellenform (5)

- Darstellung mit **`draw`**: wie zuvor ohne Monitoring
  - Monitoring nur bei Aufnahme verfügbar
  - Entsprechend kein **`void keyPressed()`** für das Ein- und Ausschalten des Monitoring erforderlich

```
void draw() {
  background(0); // schwarzer Hintergrund
  stroke(255);   // weiße Wellenlinien
  // draw the waveforms of what we are monitoring
  for(int i = 0; i < in.bufferSize() - 1; i++) {
    line( i, 50 + in.left.get(i)*50,
          i+1, 50 + in.left.get(i+1)*50);
    line( i, 150 + in.right.get(i)*50,
          i+1, 150 + in.right.get(i+1)*50 );
  }
}
```

Toninhalte visualisieren: Frequenzbereich (1)

- Tonsignale sind Gemische aus unterschiedlichen Frequenzen
- Das Frequenzspektrum eines Tonsignals lässt sich mit Hilfe der Fast Fourier Transform (FFT) bestimmen
- FFT in Sound Library und Minim vorhanden
- Hier beispielhaft für Minim:

```
import ddf.minim.analysis.*;
import ddf.minim.*;
Minim  minim;
AudioPlayer jingle;
FFT fft;
```

Toninhalte visualisieren: Frequenzbereich (2)

```
void setup() {
  size(512, 200,);
  minim = new Minim(this);
  // we want the audio buffers to be 1024 samples
  // FFT needs to have a power-of-two buffer size
  jingle = minim.loadFile("audiofile.wav", 1024);
  // loop the file indefinitely
  jingle.loop();
  fft = new FFT( jingle.bufferSize(),
                 jingle.sampleRate() );
}
```

Toninhalte visualisieren: Frequenzbereich (3)

```
void draw() {
  background(0);
  stroke(255);

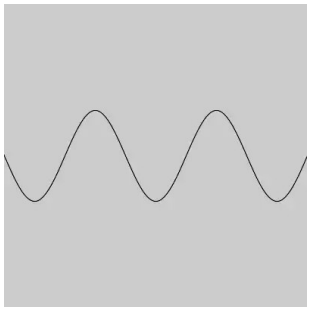
  // perform a forward FFT on the samples
  // contains the mix of both the left and right
  // channels of the file
  fft.forward( jingle.mix ); // audio => frequency

  for(int i = 0; i < fft.specSize(); i++) {
    // draw the line for frequency band i,
    // scaling it up a bit so we can see it
    line( i, height, i, height - fft.getBand(i)*8 );
  }
}
```

Toninhalte generieren: Oszillatoren

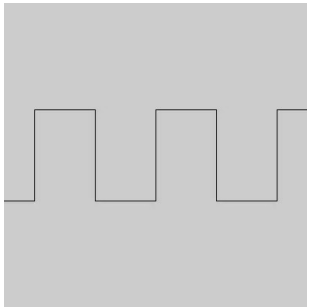
- Verfügbare Oszillatoren in der Sound Library
- Unter Minim entsprechend: Ein Oszillator mit unterschiedlichen Wellenformen

Sinusoszillator



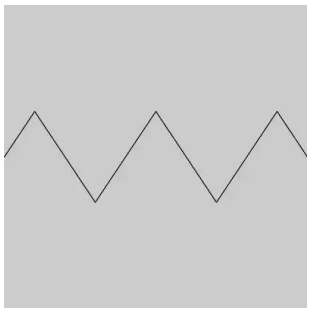
`SqrOsc`

Rechteckoszillator



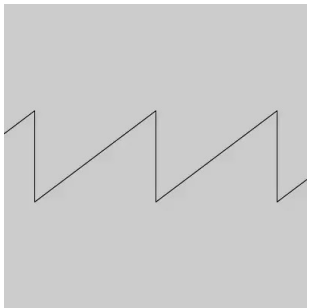
`SqrOsc`

Dreieckoszillator



`TriOsc`

Sägezahnoszillator



`SawOsc`

Toninhalte generieren: Rauschgeneratoren

- Bestandteil bei der Synthese von Klängen
  - Z. B. für perkussive Instrumente
- Arten:
  - Weißes Rauschen: Energie gleichmäßig über das gesamte Frequenzspektrum verteilt
  - Pinkes Rauschen: Energie stärker hin zu niedrigeren Frequenzen verlagert (Energieabnahme 3dB/Oktave)
  - Braunes bzw. rotes Rauschen: Energie noch stärker hin zu niedrigeren Frequenzen verlagert (Energieabnahme 6dB/Oktave)
  - Weitere: siehe z. B. https://en.wikipedia.org/wiki/Colors_of_noise
- Siehe Minim-Noise-Example: https://code.compartmental.net/minim/noise_class_noise.html

Allgemeine Methoden: Sound Library

- `amp()`: Change the amplitude/volume of this sound
- `freq()`: Sets the frequency of the oscillator
- `pan()`: Move the sound in a stereo panorama
- `play()`: Starts the oscillator
- `set()`: Set multiple parameters at once (deprecated)
- `stop()`: Stop the oscillator from playing back

Beispiel: Sinusoszillator (1)

```
import processing.sound.*;

SinOsc sine;

float amp=0.5;
float add=0.0;
float pos=0;

void setup() {
  // Create the sine oscillator.
  sine = new SinOsc(this);
  sine.play();
}
```

Beispiel: Sound Lib.: Sinusoszillator (2)

```
void keyReleased() {
  switch(key) {
    case '1':
      sine.play(100, amp, add, pos);
      break;
    case '2':
      sine.play(500, amp, add, pos);
      break;
    case '3':
      sine.play(1000, amp, add, pos);
      break;
    case '4':
      sine.play(1500, amp, add, pos);
}}
```

Beispiel: Minim-Wellengenerator (1)

```
import ddf.minim.*;
import ddf.minim.ugens.*;
import ddf.minim.analysis.*;

Minim       minim;
AudioOutput out;
Oscil       wave;

void setup() {
  size(512, 200, P3D);
  minim = new Minim(this);

  // use getLineOut method of Minim object to get
  out = minim.getLineOut(); // an AudioOutput object

  // create a sine wave Oscil, set to 440 Hz, at 0.5 amplitude
  wave = new Oscil( 440, 0.5f, Waves.SINE );

  // patch the Oscil to the output
  wave.patch( out );
}
```

Beispiel: Minim-Wellengenerator (2)

```
void draw() {
  background(0); stroke(255); strokeWeight(1);

  // draw the waveform of the output
  for(int i = 0; i < out.bufferSize() - 1; i++) {
    line( i, 50  - out.left.get(i)*50,
          i+1, 50  - out.left.get(i+1)*50 );
    line( i, 150 - out.right.get(i)*50,
          i+1, 150 - out.right.get(i+1)*50 );
  }

  // draw the waveform we are using in the oscillator
  stroke( 128, 0, 0 ); // color: red
  strokeWeight(4);
  for( int i = 0; i < width-1; ++i ) {
    point( i, height/2 - (height*0.49) *
           wave.getWaveform().value( (float)i / width ) );
  }
}
```

Beispiel: Minim-Wellengenerator (3)

- Frequenz und Lautstärke mit Mausposition steuern:

```
void mouseMoved() {

  // but this is a quick and easy way to turn the screen
  // into an x-y control for amplitude and frequency.

  float amp = map( mouseY, 0, height, 1, 0 );
  wave.setAmplitude( amp );

  float freq = map( mouseX, 0, width, 110, 880 );
  wave.setFrequency( freq );
}
```

Beispiel: Minim-Wellengenerator (4)

- Wellenform mit Tastatur wählen:

```
void keyPressed() {
  switch( key ) {
    case '1': wave.setWaveform( Waves.SINE ); break;

    case '2': wave.setWaveform( Waves.TRIANGLE ); break;

    case '3': wave.setWaveform( Waves.SAW ); break;

    case '4': wave.setWaveform( Waves.SQUARE ); break;

    case '5': wave.setWaveform( Waves.QUARTERPULSE ); break;
  }
}
```

Literatur

[1] Eckel, Bruce: *Thinking in Java*, Prentice Hall, 2003

[2] Gallardo, R., et al.: *The Java Tutorial*: A Short Course on the Basics, https://docs.oracle.com/javase/tutorial/, 17.3.2020

[3] Gumm, H.-P.; Sommer, M.: *Einführung in die Informatik*, Oldenbourg, 2002

[4] Kjell, Bradley: *Introduction to Computer Science using Java*, 2014

[5] Processing Documentation: https://processing.org/reference/