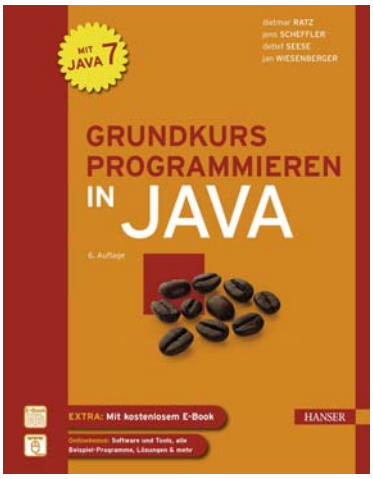
HANSER



Erratum

zu

„Grundkurs Programmieren in Java“

(6. Auflage)

von Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

ISBN (Buch): 978-3-446-42663-4
ISBN (E-Book): 978-3-446-42835-5

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-42663-4
sowie im Buchhandel

Liebe Leser,

leider haben sich im Buch ein paar Fehler eingeschlichen.
Auf den nächsten Seiten finden Sie den korrigierten Text
(gelb markiert).

Wenn Sie Anregungen oder Fragen haben, können Sie gerne auch über die folgende Website

http://www.grundkurs-java.de

mit den Autoren Kontakt aufnehmen.

Wir danken für Ihr Interesse.

In Zeile 3 der Datei `Longtst.java` haben wir also eine Zahl verwendet, die zu groß ist. Zu groß deshalb, weil sie ja standardmäßig als **`int`**-Wert angenommen wird, aber laut Tabelle 4.2 deutlich größer als `2147483647` ist und somit nicht mehr mit 32 Bits dargestellt werden kann. Eine derartige Zahl muss explizit als längere Zahl gekennzeichnet werden! Wir ändern die Zeile deshalb wie folgt:

```
System.out.println(9223372036854775807L);
```

Durch die Hinzunahme der Endung `L` wird die Zahl als eine **`long`**-Zahl betrachtet und somit mit 64 Bits codiert (in die sie laut Tabelle gerade noch hineinpasst). Der entsprechende Datentyp heißt in Java **`long`**.
Achtung: Neben der rein dezimalen Schreibweise von ganzzahligen Werten können Literalkonstanten in Java auch als oktale Zahlen (Zahlen im Achter-System) oder als hexadezimale Zahlen (Zahlen im Sechzehner-System) geschrieben werden. Oktale Zahlen müssen mit einer führenden `0` beginnen und dürfen nur Ziffern im Bereich 0 bis 7 enthalten. Hexadezimale Zahlen müssen mit `0x` beginnen und dürfen Ziffern im Bereich 0 bis 9 und A bis F enthalten. Die drei **`int`**-Konstanten `27`, `033` und `0x1B` sind somit alternative Schreibweisen für den ganzzahligen dezimalen Wert 27.

4.3.2 Gleitkommatypen

Wir wollen eine einfache Rechung durchführen. Das folgende Programm soll das Ergebnis von `1/10` ausgeben und verwendet dazu den Divisionsoperator in Java:

```
1 public class Intdiv {
2   public static void main (String[] args) {
3     System.out.println(1/10);
4   }
5 }
```

Wir übersetzen das Programm und führen es aus. Zu unserer Überraschung erhalten wir jedoch ein vermeintlich falsches Ergebnis – und zwar die Null! Was ist geschehen?
Um zu begreifen, was eigentlich passiert ist, müssen wir uns eines klar machen: wir haben mit ganzzahligen Datentypen gearbeitet. Der Divisionsoperator ist in Java jedoch so definiert, dass die Division zweier ganzer Zahlen wiederum eine ganze Zahl (nämlich den ganzzahligen Anteil des Quotienten) ergibt. Wir erinnern uns an die Grundschulzeit – hier hätte `1/10` ebenfalls 0 ergeben – mit Rest 1. Diesen Rest können wir in Java mit dem `%`-Zeichen bestimmen. Wir ändern unser Programm entsprechend:

```
1 public class Intdiv {
2   public static void main (String[] args) {
3     System.out.print("1/10 betraegt ");
4     System.out.print(1/10);  // ganzzahliger Anteil
5     System.out.print(" mit Rest ");
6     System.out.print(1%10);  // Rest
7   }
8 }
```

```
60            default: // Falsche Zahl eingegeben
61              System.out.println("Eingabefehler!");
62          }
63        } // Schleifenende
64      } // Ende des Hauptprogramms
65    } // Ende des Programms
```

Unsere neuen Objekte `adr0` und `adr1` werden hierbei in den Zeilen 17 und 18 vereinbart.[7] Die Referenz `adr` wird anfangs auf `adr0` gesetzt (Zeile 19). Das Menü wird um einen zusätzlichen Eintrag erweitert (Zeile 31), der in den Zeilen 52 bis 56 implementiert wird. Das restliche Programm stimmt mit dem von Seite 137 überein.

5.2.6 Felder von Klassen

Wir haben im letzten Abschnitt gelernt, dass wir mit nur wenig Mehraufwand unser Adressprogramm von der Verwaltung *einer* Adresse auf die Verwaltung von mehr als einer Adresse ausweiten konnten: durch die Verwendung von Referenzen konnten wir das allgemeine Problem *mehrerer* Adressen (`adr0` und `adr1`) auf die Verwaltung *einer* Adresse (`adr`) zurückführen. Wie können wir unser Programm nun so modifizieren, dass wir auch eine größere Zahl von Datensätzen verarbeiten?
Der Gedanke liegt nahe, hierfür auf unser Wissen aus Abschnitt 5.1 zurückzugreifen. Wir haben eine Vielzahl von Daten (z. B. zwanzig Adresseinträge), die wir etwa in Form einer Tabelle anordnen könnten. Wir nummerieren die Adresseinträge von 0 bis 19 durch und speichern sie in einem Feld namens `adressen`:

```
Adresse[] adressen= new Adresse[20];
```

Dieser einfache Ansatz führt tatsächlich bereits zu dem gewünschten Ergebnis. Es ist nämlich so, dass in Java Felder nicht nur über einfachen Datentypen (**`int`**, **`double`**,...), sondern auch über Referenzdatentypen aufgebaut werden können. Hierzu zählen sowohl Felder (siehe Abschnitt 5.1.6 und 5.1.7) als auch sonstige Klassen. Wir haben diesen Umstand unbewusst schon genutzt, indem wir für unseren Terminkalender Felder über Strings bildeten. Nun wollen wir uns diese Eigenschaft von Java jedoch auch bewusst zu Nutze machen.
Abbildung 5.16 zeigt den Zustand unseres Feldes `adressen` nach der Erzeugung mit Hilfe des **`new`**-Operators. Die Variable `adressen` umfasst insgesamt zwanzig Einträge, das heißt, sie verweist auf die Komponenten `adressen[0]` bis `adressen[19]`, die vom Typ `Adresse` sind. Da es sich hierbei um eine benutzerdefinierte Klasse handelt, also auch um einen Referenzdatentyp, stellt jeder dieser Einträge wieder eine Referenz dar. Diese zeigt zu Anfang „nirgendwohin“, d. h. sie referenziert kein spezifisches Objekt. Es handelt sich um die so genannte **Null-Referenz**, die in Java mit **`null`** bezeichnet wird.

[7] Den aufmerksamen Lesern wird hier wahrscheinlich etwas aufgefallen sein: Warum werden die Objekte hier als `adr0` und `adr1` bezeichnet, obwohl es sich doch nur um die Namen der *Referenzen* handelt? Tatsächlich „heißen“ die Objekte natürlich nicht `adr0` oder `adr1`; aus Gründen der Übersichtlichkeit wird diese Ungenauigkeit aber üblicherweise in Kauf genommen.

Kapitel 6

Methoden, Unterprogramme

Wir wollen ein einfaches Problem lösen: Für die Funktion $f(x,n) = x^{2n} + n^2 - nx$ mit positivem ganzzahligem n und reellem x sind Funktionswerte zu berechnen. Folgendes Programm tut genau dies:

```
1  import Prog1Tools.IOTools;
2  public class Eval1 {
3    public static void main(String[] args) {  // Hauptprogramm
4      int n = IOTools.readInteger("n=");       // lies n ein
5      double x = IOTools.readDouble("x=");     // lies x ein
6      double produkt = 1.0;                    // Berechnung der Potenz
7      for (int i=0; i < 2*n; i++)              // ...
8        produkt = produkt * x;                 // abgeschlossen
9      double f_x_n = produkt + n*n - n*x;      // Berechnung von f
10     System.out.println("f(x,n)=" + f_x_n);  // Ergebnisausgabe
11   }
12 }
```

Nun wollen wir das Problem etwas komplizieren. Statt eines einfachen x soll man einen Bereich angeben können – ein Intervall, in dem $f(x,n)$ wie folgt ausgewertet wird:

1. Werte f am linken Randpunkt `l` des Intervalls aus.
2. Werte f am rechten Randpunkt `r` des Intervalls aus.
3. Werte f am Mittelpunkt des Intervalls (berechnet aus `(l+r)/2`) aus.
4. Gib den Mittelwert der drei Funktionswerte aus.

Wir erweitern unser Programm entsprechend. Hierbei definieren wir für die drei Auswertungen der Funktion jeweils drei eigenständige Variablen, um sie für den späteren Gebrauch zu speichern. Das entstandene Programm sieht nun so aus:

```
1  import Prog1Tools.IOTools;
2  public class Eval2 {
3    public static void main(String[] args) {  // Hauptprogramm
4      int n = IOTools.readInteger("n=");      // lies n ein
5      double l = IOTools.readDouble("l=");    // lies l ein
6      double r = IOTools.readDouble("r=");    // lies r ein
7
8      double produkt = 1.0;                   // Berechnung der Potenz
9      for (int i=0; i < 2*n; i++)             // ...
10       produkt = produkt * l;                // abgeschlossen
11     double f_l_n = produkt + n*n - n*l;     // Berechnung von f
12     System.out.println("f(l,n)=" + f_l_n);  // Ergebnisausgabe
13
14     produkt = 1.0;                          // Berechnung der Potenz
15     for (int i=0; i < 2*n; i++)             // ...
16       produkt = produkt * r;                // berechnet
17     double f_r_n = produkt + n*n - n*r;     // Berechnung von f
18     System.out.println("f(r,n)=" + f_r_n);  // Ergebnisausgabe
19
20     double m = (l + r) / 2.0;               // Mittelpunkt
21     produkt = 1.0;                          // Berechnung der Potenz
22     for (int i=0; i < 2*n; i++)             // ...
23       produkt = produkt * m;                // abgeschlossen
24     double f_m_n = produkt + n*n - n*m;     // Berechnung von f(M,n)
25     System.out.println("f(m,n)=" + f_m_n);  // Ergebnisausgabe
26
27     double mw = (f_l_n + f_r_n + f_m_n)/3;  // Mittelwert
28     System.out.println("Mittelwert=" + mw); // Ergebnisausgabe
29   }
30 }
```

Wir sehen, dass unser neues Programm wesentlich länger und leider auch unübersichtlich geworden ist. Der Grund hierfür liegt vor allem an der sich ständig wiederholenden **`for`**-Schleife. Leider benötigen wir diese aber für die Berechnung der Funktion f. Zu schade, dass wir diese nicht wie den Sinus oder Tangens als einen eigenständigen Befehl zur Verfügung stellen können! Oder etwa doch?
In den folgenden Abschnitten lernen wir, so genannte **Methoden** (oder auch **Routinen**) zu definieren. Dies sind Unterprogramme, die vom Hauptprogramm (der `main`-Methode) aufgerufen werden und auch Ergebnisse zurückliefern können. Mit ihrer Hilfe werden wir Programme schreiben, die weit komplexer als obiges Beispiel, aber dennoch übersichtlicher sind!

6.1 Methoden

6.1.1 Was sind Methoden?

Durch Methoden wird ausführbarer Code unter einem Namen zusammengefasst. Dieser Code kann unter Verwendung so genannter Parameter formuliert sein, denen später beim Aufruf der Methode Werte übergeben werden. Wie im Abschnitt über Klassen bereits erwähnt, gehören Methoden in der Regel neben den Varia-

Parameterliste besteht aus einem Feld von Strings, das den Namen `args` trägt, was wir aber auch `nasenbaer` oder `schoenesFeld` hätten nennen können, weil nur der Typ, nicht aber der Name des Parameters relevant ist.

6.1.3 Parameterübergabe und Ergebnisrückgabe

Wir wollen nun unsere Funktion $f(x,n) = x^{2n} + n^2 - nx$ durch eine Methode berechnen lassen. Wie haben wir diese zu programmieren?
Als Erstes müssen wir uns Gedanken über den Kopf der Methode machen. Welchen Rückgabetyp hat die Methode? Welche Parameter müssen wir übergeben? Und wie sollen wir sie nur benennen?
Letztgenanntes Problem dürfte relativ schnell gelöst sein – wir nennen sie einfach `f`. Dies ist schließlich der Name der Funktion, und es handelt sich hierbei um einen Bezeichner, der kein reserviertes Wort darstellt. Auch der Rückgabetyp ist relativ leicht geklärt. Wir haben in unserem Programm Gleitkommawerte stets durch **`double`**-Zahlen codiert und werden dies auch weiterhin tun. Als Rückgabetyp legen wir deshalb einfach **`double`** fest.
Bezüglich der Parameterliste haben wir zwei Werte, die wir der Funktion übergeben müssen:

- einen ganzzahligen Wert `n`, den wir im Hauptprogramm in einer Variable vom Typ **`int`** abgespeichert hatten und
- eine Gleitkommazahl `x`, die wir durch einen **`double`**-Wert codieren.

Wir haben somit alle Informationen zusammen, um unseren Methodenkopf zu definieren. Dieser lautet nun wie folgt:

```
public static double f(double x, int n) {
```

Wie wir nun den Funktionswert $f(x,n)$ berechnen, ist klar: auf die gleiche Weise wie in den bisherigen Programmen. Ein entsprechendes Codestück könnte etwa so aussehen:

```
double produkt = 1.0;                    // Berechnung der
for (int i=0; i < 2*n; i++)              // Potenz x^2n
  produkt = produkt * x;                 // abgeschlossen
double ergebnis = produkt + n*n - n*x;   // Berechnung von f(x,n)
```

Wie machen wir Java jedoch klar, dass in der Variable `ergebnis` nun das Ergebnis unserer Methode steht? Wie erkennt das Programm, dass als Ergebnis nicht etwa `produkt` zurückgegeben werden soll? Für diese Ergebnisrückgabe an die aufrufende Umgebung steht das Kommando **`return`** zur Verfügung.
Durch den Befehl

```
return ergebnis;
```

wird die Ausführung der Methode beendet und der Inhalt der Variable `ergebnis` als Resultat zurückgegeben. Die Variable muss natürlich vom gleichen Typ wie der Rückgabetyp oder durch implizite Typumwandlung in den entsprechenden Typ umwandelbar sein.

Auch wenn wir bislang noch nicht über das sprachliche Wissen verfügen, um diese Zeilen vollständig zu verstehen (der Vererbung ist mit Kapitel 9 ein eigener Teil dieses Buches gewidmet), so lässt sich doch relativ einfach nachvollziehen, was sie zu bedeuten haben:

- Die Klasse `FaxAdresse` stellt eine Subklasse von `Adresse` dar. Sie spezialisiert oder *erweitert* also die eigentliche Adresse – im Englischen erklärt dies das Schlüsselwort **`extends`**.
- Als Subklasse von `Adresse` *ist* eine `FaxAdresse` automatisch auch eine `Adresse`. Sie *erbt* sämtliche Eigenschaften ihrer Superklasse, sodass wir die Instanzvariablen `name` oder `hausnummer` nicht erneut definieren müssen. Sie sind dank der verwandtschaftlichen Beziehung automatisch vorhanden!
- Da der größte Teil unserer Arbeit bereits mit der Klasse `Adresse` erledigt wurde, können wir uns auf jene neuen Aspekte beschränken, die wir unserer Subklasse hinzufügen wollen. In diesem Fall bedeutet dies die Definition zweier neuer Variablen in den Zeilen 3 und 4.

Wie wir sehen, kann uns die Kombination von Generalisierung und Vererbung in der Programmierung eine Menge Schreibarbeit ersparen. Dies ist jedoch nicht der einzige Vorteil, den uns diese beiden Grundpfeiler der Objektorientierung bieten:

- Weil wir durch Vererbung gemeinsame Eigenschaften nur einmal modellieren müssen, brauchen wir diese Eigenschaften auch nur an einer Stelle zu testen. Wir haben nur eine Möglichkeit, Programmierfehler einzubauen, und somit auch nur eine Stelle, an der wir diese korrigieren müssen. Würden wir etwa den Vorgang des Milchgebens bei jedem Tier einzeln realisieren, so müssten wir jede dieser neu geschriebenen Methoden auf Fehler überprüfen.
- Algorithmen, die gewisse Spezialeigenschaften einer Subklasse nicht benötigen, können für die allgemeinere Superklasse definiert werden. Auf diese Weise können sie auch automatisch auf die verschiedensten Subklassen angewendet werden. So kann Java beispielsweise alle Objekte sortieren, die sich auf eine bestimmte Art und Weise miteinander vergleichen lassen („größer als“, „kleiner als“). Wollten wir also unsere Adressen sortieren, müssten wir lediglich dafür sorgen, dass sich unsere Objekte auf die richtige Art und Weise miteinander vergleichen lassen – den Rest erledigt eine vordefinierte Methode.

Vererbung hilft Programmierern also nicht nur, Fehler zu vermeiden. Sie erlaubt es den Entwicklern auch, sich im Laufe der Zeit ganze Bibliotheken von vorgefertigten Objekten für jeden Zweck zusammenzustellen, die sie bei Bedarf einfach um zusätzliche Funktionalität erweitern. Auf diese Weise lassen sich Softwareprodukte schneller und günstiger auf den Markt bringen als mit konventionellen Programmiersprachen.

```
1  public class TestStrecke {
2    public static void main(String[] args) {
3      Punkt ursprung = new Punkt(0.0,0.0);
4      Punkt endpunkt = new Punkt(4.0,3.0);
5      Strecke s = new Strecke(ursprung,endpunkt);
6      System.out.println("Die Laenge der Strecke " + s +
7                         " betraegt " + s.getLaenge() + ".");
8      System.out.println();
9      System.out.println("Strecke s eingeben:");
10     s.read();
11     System.out.println();
12     System.out.println("Die Laenge der Strecke " + s +
13                        " betraegt " + s.getLaenge() + ".");
14   }
15 }
```

Aufgabe 8.11

Gegeben sei die folgende Klasse:

```
1  public class AchJa {
2
3    public int x;
4    static int ach;
5
6    int ja (int i, int j) {
7      int y;
8      if ((i <= 0) || (j <= 0) || (i % j == 0) || (j % i == 0)) {
9         System.out.print(i+j);
10        return i + j;
11     }
12     else {
13        x = ja(i-2,j);
14        System.out.print(" + ");
15        y = ja(i,j-2);
16        return x + y;
17     }
18   }
19
20   public static void main (String[] args) {
21     int n = 5, k = 2;
22     AchJa so = new AchJa();
23     System.out.print("ja(" + n + "," + k + ") = ");
24     ach = so.ja(n,k);
25     System.out.println(" = " + ach);
26   }
27 }
```

a) Geben Sie an, um welche Art von Variablen es sich bei den in dieser Klasse verwendeten Variablen `x` in Zeile 3, `ach` in Zeile 4, `j` in Zeile 6, `y` in Zeile 7, `n` in Zeile 21 und `so` in Zeile 22 jeweils handelt. Verwenden Sie (sofern diese zutreffen) die Bezeichnungen Klassen-Variable, Instanz-Variable, lokale Variable und formale Variable (bzw. formaler Parameter).

b) Geben Sie an, was das Programm ausgibt.

c) Angenommen, die Zeile 24 würde in der Form

```
ach = ja(n,k);
```

gegeben sein. Würde der Compiler das Programm trotzdem übersetzen? Wenn nein, warum nicht?

Aufgabe 8.12

Es sei folgende einfache Klasse gegeben, die zur Speicherung von Daten über Patienten in der Aufnahme einer Arztpraxis verwendet werden könnte.

```
1 public class Patient {
2   public String name;          // Name des Patienten
3   public int alter;            // Alter (in Jahren)
4   public int altersDifferenz (int alter) {
5     return Math.abs(alter - this.alter);
6   }
7 }
```

a) Erläutern Sie den Aufbau der Klasse grafisch.

b) Was passiert durch die nachfolgenden Anweisungen?

```
Patient maier;
maier = new Patient();
```

c) Wie würde ein geeigneter Konstruktor für die Klasse `Patient` aussehen? Wenn Sie die Klasse um diesen Konstruktor ergänzen, ist dann die Anweisung

```
Patient maier = new Patient();
```

noch zulässig?

d) Erweitern Sie die Klasse `Patient` um eine Instanzvariable `vorherDran`, die eine Referenz auf einen weiteren Patienten darstellt, und um eine Instanzvariable `nummer`, die es ermöglicht, allen Patienten (z. B. bei der Erzeugung eines neuen Objektes für eine Warteliste einer Praxis) eine (eindeutige) ganzzahlige Nummer zuzuordnen.

e) Erweitern Sie die Klasse `Patient` um eine Klassenvariable `folgeNummer`, die die jeweils nächste zu vergebende Nummer enthält.

f) Modifizieren Sie den Konstruktor der Klasse `Patient` so, dass er jeweils eine entsprechende Nummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, auch noch den Vorgänger in der Warteliste anzugeben.

g) Wie verändert sich der Wert der Variablen `nummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

3. Geben wir die Variable `s.var` direkt aus, so erhalten wir wie erwartet erneut die 2 als Ergebnis.

4. Führen wir jedoch eine Umwandlung der Referenz in den Typ `Vater` durch, so erhalten wir völlig überraschend als Ergebnis die 1. Das liegt nun daran, dass für Variablen keine dynamische Bindung ins Spiel kommt, sondern der Compiler bereits zur Übersetzungszeit abhängig vom Typ der Referenz entscheidet, auf welche der Variablen zugegriffen wird.

Was wollte uns dieser Abschnitt also sagen? Das Prinzip des Polymorphismus bzw. das dynamische Binden greift lediglich bei Methoden, *nicht* bei Variablen.

9.3.2 Überschreiben von Methoden verhindern durch `final`

Wollen wir dafür sorgen, dass eine Methode in allen Unterklassen in der gleichen Version vorliegt, die Polymorphie also ausschalten, können wir wiederum das Schlüsselwort **`final`** einsetzen. Ähnlich der Vorgehensweise bei der Markierung von ganzen Klassen, die nicht mehr erweitert werden können, verhindert der Modifizierer **`final`** vor einer Methoden-Deklaration das Überschreiben dieser Methode in einer Unterklasse. Definieren wir also eine Klasse

```
1 public class Papa {
2   public final void singe() {
3     System.out.println("La la la la la ...");
4   }
5 }
```

und versuchen anschließend in der Klasse

```
1 public class Kind extends Papa {
2   public void singe() {
3     System.out.println("Do Re Mi Fa So ...");
4   }
5 }
```

die Methode `singe` zu überschreiben, erhalten wir beim Compilieren

Konsole

```
Kind.java:2: singe() in Kind cannot override singe() in Papa;
  overridden method is final
    public void singe() {
                ^
```

als Fehlermeldung vom Compiler. Auf einige Beispiele solcher finalen Methoden kommen wir in Kapitel 18 noch zu sprechen. Sie finden sich in der Klasse `Object`, auf die wir im folgenden Abschnitt eingehen. Durch die **`final`**-Deklaration dieser Methoden ist sichergestellt, dass bei diesen Methoden alle Java-Objekte das gleiche Verhalten aufweisen. Außerdem ist der entsprechende compilierte Programmcode effizienter, weil kein dynamisches Binden mehr durchgeführt werden muss.

14.4.3.1 Die Klasse `FlowLayout`

Um die Komponenten in einem Container fließend anzuordnen, verwendet man ein Objekt der Klasse `FlowLayout` als Layout-Manager. „Fließend“ bedeutet hier, dass die Komponenten zeilenweise von links nach rechts in den Container eingefügt werden. Das heißt, die Komponenten werden so lange in der Reihenfolge ihres Einfügens von links nach rechts nebeneinander platziert, bis kein Platz mehr für die nächste Komponente verfügbar ist und mit einer neuen Zeile begonnen werden muss, die dann genauso gefüllt wird. Die Ausrichtung der Komponenten innerhalb der Zeile erfolgt dabei standardmäßig zentriert. Zwischen den Komponenten findet sich horizontal und vertikal jeweils ein Abstand von 5 Pixel. Die Größe der Komponenten wird nicht verändert.
In der Klasse `FlowLayout` finden wir die folgenden Konstruktoren:

- **`public`** `FlowLayout()`
  erzeugt ein `FlowLayout`-Objekt mit den Standardeinstellungen (zentrierte Ausrichtung der Zeilen, 5-Pixel-Abstände).
- **`public`** `FlowLayout(`**`int`** `align)`
  erzeugt ein `FlowLayout`-Objekt mit einer Ausrichtung gemäß `align` und der Standardeinstellung für die Abstände.
- **`public`** `FlowLayout(`**`int`** `align,` **`int`** `h,` **`int`** `v)`
  erzeugt ein `FlowLayout`-Objekt mit einer Ausrichtung gemäß `align` und horizontalen bzw. vertikalen Abständen von `h` bzw. `v` Pixel.

Für die Wahl der Ausrichtung stehen die vordefinierten Konstanten `LEFT` (für linksbündige Ausrichtung), `RIGHT` (für rechtsbündige Ausrichtung) und `CENTER` (für zentrierte Ausrichtung) als finale Klassenvariablen der Klasse `FlowLayout` zur Verfügung.
In unserem Beispielprogramm

```
1  import java.awt.*;
2  import javax.swing.*;
3  /** Erzeuge ein Swing-Fenster mit Flowlayout */
4  public class FrameMitFlowLayout extends JFrame {
5    Container c;            // Container dieses Frames
6    // Feld fuer Labels, die im Frame erscheinen sollen
7    FarbigesLabel fl[] = new FarbigesLabel[4];
8
9    public FrameMitFlowLayout() {    // Konstruktor
10     c = getContentPane();             // Container bestimmen
11     c.setLayout(new FlowLayout());    // Layout setzen
12
13     // Erzeuge die Labelobjekte mit Uebergabe der Labeltexte
14     for (int i = 0; i < 4; i++) {
15       int rgbFg = 255 - i*80;    // Farbwert fuer Vordergrund
16       int rgbBg = i*80;          // Farbwert fuer Hintergrund
17       fl[i] = new FarbigesLabel("Nummer " + (i+1),
18                                 new Color(rgbFg,rgbFg,rgbFg),
19                                 new Color(rgbBg,rgbBg,rgbBg));
20       fl[i].setFont(new Font("Serif",Font.ITALIC,28));
```

die Werkzeugleiste in einem eigenen Fenster mit Titelleiste dargestellt. Man kann diese Fähigkeit auch unterbinden, indem man einen Aufruf der Instanzmethode `setFloatable(false)` verwendet.
Im folgenden Beispielprogramm, das die Vorstufe zu einem einfachen „Wechselbilderrahmen“ bildet, haben wir unsere Oberflächen-Komponenten teilweise in ein Menü der Menüleiste und teilweise in eine Werkzeugleiste gepackt.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  /** Erzeuge ein einfaches Swing-Fenster mit einem Menue einer
5      Toolbar und einem Textlabel */
6  public class FrameMitMenuBar extends JFrame {
7    Container c;           // Container dieses Frames
8    JMenuBar menuBar;      // Menueleiste
9    JMenu menu;            // Menue
10   JMenuItem menuItem;    // Menue-Eintrag
11   JToolBar toolBar;      // Werkzeugleiste
12   JButton button;        // Knoepfe der Werkzeugleiste
13   JLabel textLabel;      // Label, das im Frame erscheinen soll
14
15   public FrameMitMenuBar() {  // Konstruktor
16     // Bestimme die Referenz auf den eigenen Container
17     c = getContentPane();
18
19     // Erzeuge die Menueleiste.
20     menuBar = new JMenuBar();
21     // Erzeuge ein Menue
22     menu = new JMenu("Bilder");
23     menu.setMnemonic(KeyEvent.VK_B);
24     // Erzeuge die Menue-Eintraege und fuege sie dem Menue hinzu
25     menuItem = new JMenuItem("Hund");
26     menuItem.setMnemonic(java.awt.event.KeyEvent.VK_H);
27     menu.add(menuItem);
28     menuItem = new JMenuItem("Katze");
29     menuItem.setMnemonic(java.awt.event.KeyEvent.VK_K);
30     menu.add(menuItem);
31     menuItem = new JMenuItem("Maus");
32     menuItem.setMnemonic(java.awt.event.KeyEvent.VK_M);
33     menu.add(menuItem);
34     // Fuege das Menue der Menueleiste hinzu
35     menuBar.add(menu);
36     // Fuege die Menueleiste dem Frame hinzu
37     setJMenuBar(menuBar);
38
39     // Erzeuge die Werkzeugleiste
40     toolBar = new JToolBar("Rahmenfarbe");
41     // Erzeuge die Knoepfe
42     button = new JButton(new ImageIcon("images/rot.gif"));
43     button.setToolTipText("roter Rahmen");
44     toolBar.add(button);
45     button = new JButton(new ImageIcon("images/gruen.gif"));
46     button.setToolTipText("gruener Rahmen");
47     toolBar.add(button);
48     button = new JButton(new ImageIcon("images/blau.gif"));
```