Java

Grundlagen

mit

NetBeans

Version 2019.2

Autor: Dieter Lindenberg

Inhalt

GUI (Graphical User Interface)

In einem Java-Programm werden sehr oft (aber nicht immer) unterschiedliche, selbst definierte **Klassen** benötigt. All diese „zusammen gehörenden“ Klassen fasst man zu einem **Projekt** zusammen.

Zu Beginn unserer Java-Ausbildung werden wir viele verschiedene, einzelne Programmieraufgaben lösen. Für diese Aufgaben erstellen wir immer ein sog. **Formblatt** (das ist ein Fenster oder Rahmen, welches auf dem Bildschirm erscheint), das die für die jeweilige Aufgabe benötigten Schalter, Bilder und Textfelder enthält. Dieses Formblatt ist jeweils immer ebenfalls eine eigene Klasse (genauer: ein Objekt einer eigenen Klasse). Viele dieser Aufgaben (also Formblätter) fassen wir unter einem einzigen **Projekt** zusammen. Dieses Projekt enthält also zu Beginn unserer Java-Ausbildung viele unterschiedliche Formblätter (Aufgaben), von denen kein einziges vor den anderen bevorzugt wird.

Starte *NetBeans*, wähle oben links *File/ New Project.* Im daraufhin erscheinenden Fenster wähle unter *Categories* die Option *Java* und unter *Projects* die Option *Java Application*. Im nächsten Fenster gib dem Projekt den Namen *MeineAufgaben* und entferne das Häkchen vor *Create Main Class* (es gibt nämlich keine bevorzugte Aufgabe in diesem Projekt). Speichere dein Projekt!

Klicke im Projekte-Fenster oben links innerhalb deines jetzt neuen Projektes namens „MeineAufgaben“ mit der rechten Maustaste auf *Source packages* und wähle *New/JFrameForm*. Wähle als Klassenname *Uebung1* !

Hinweis zum NetBeans-Editor:
Wenn man Programmtexte aus einer Word- oder PDF-Datei kopiert und in den NetBeans-Editor einfügt, so wird von der GUI dieser kopierte Text üblicherweise ohne strukturierte Formatierung oder aber mit völlig unsinniger Formatierung übernommen.
Diese unsinnige Formatierung lässt sich automatisch größtenteils korrigieren, indem man nach dem Einfügen des Textes die Tastenkombination <Alt> + <Shift> + <F> betätigt.

Außerdem wird der übernommene Text (auch noch nach der automatischen Korrektur) noch als fehlerhaft beanstandet. Der Grund dafür liegt darin, dass einzelne Zeichen im PDF-Original anders dargestellt werden bzw. andere Zeichen sind (z.B. die Anführungszeichen) als im Editor verlangt werden.

Des Weiteren meldet der NetBeans-Editor auch unverständliche Fehler an Stellen, wenn weiter unten im Quellcode Fehler auftreten (z.B. überflüssige geschweifte Klammern).

Hat man eine Klasse vom Typ *JFrameForm* erzeugt, geht nach kurzer Zeit automatisch der sog. *GUI-Builder* (**G**raphical **U**ser **I**nterface) auf. Man sieht u.a. ein Formblatt. Wenn man darauf klickt, erscheint unten rechts ein Fenster mit den Eigenschaften dieses Formblattes. Dort kann man dem Formblatt z.B. einen Titel geben oder unter *preferredSize* eine entsprechende Größe geben. Die ersten beiden Koordinaten bestimmen den Abstand des Formblattes vom Bildschirmrand.

Die für uns zunächst wichtigsten Elemente im GUI sind:

Label	einzeilige Textausgabe
Textfield	einzeilige Textein/ausgabe
Textarea	mehrzeilige Textein/ausgabe
Button	Schaltknopf

Ziehe ein *Text Field* auf das Formblatt! Wenn man anschließend mit der rechten Maustaste auf dieses Textfeld klickt, lässt sich unter *Edit Text* der Inhalt festlegen und unter *Change Variable Name* kann man den Namen dieses Textfeldes bestimmen. Nenne es *tf1* ! Unter *properties* lassen sich verschiedene Eigenschaften des Textfeldes einstellen. Wähle eine Hintergrundfarbe und eine Vordergrundfarbe, stelle die Schriftart und Schriftgröße ein! Wähle die Eigenschaft *Zentriert*!

Ziehe einen Button auf das Formblatt!
Analog wie bei Textfeldern kann man Beschriftung, Variablennamen und weitere Eigenschaften des Buttons einstellen. Nenne ihn *btStart*!

Nach einem Rechtsklick auf den Button kann man ein *Event* (Ereignis) auswählen. Wähle *MouseClicked*. Sofort wird automatisch vom Design-Fenster in das Source-Fenster gewechselt. Dort sieht man schon vorgefertigte Programmteile. Schreibe in die entsprechende Methode
private void btStartMouseClicked(java.awt.event.MouseEvent evt)
noch folgende Anweisung: **`tf1.setText("Informatik ist toll");`**

Das Programm wird am besten dadurch gestartet, indem man links im Projektfenster mit der rechten Maustaste auf den gerade erstellten Klassennamen klickt und *Run File* auswählt.

Button, Label, Text-Field

Ein ***Button*** *(deutsch: Knopf; in der Informatik ist damit eine Schaltfläche gemeint)* besitzt viele Methoden und Eigenschaften. Zwei davon sind für die Schule interessant:

```
public   void setVisible(boolean b)

public   void setText(String s)
```

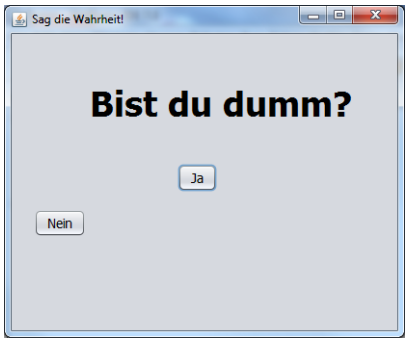
Ein ***Label*** ist ein Text, der üblicherweise auf dem Formblatt steht. Dieser Text kann vom Anwender des Programmes nicht überschrieben werden. Ein *Label* besitzt ebenfalls Dutzende von Methoden und Eigenschaften. Interessant für uns wären hauptsächlich nur die folgenden Methoden:

```
public   void setText(String  s)
         // das bedeutet, der Programmierer bzw. das Programm selbst kann
            den Text überschreiben.

public   void setVisible(boolean b)
```

In einem ***Text Field*** kann üblicherweise auch der Anwender Texte eingeben. Es besitzt ebenfalls Dutzende von Methoden und Eigenschaften. Interessant für uns wären hauptsächlich nur die folgenden Methoden:

```
public   void setText(String s)

public   String getText()

public   void requestFocus()
         // Danach besitzt das Textfeld den Fokus.

public   void setEditable(Boolean b)
         // Entscheidet, ob der Benutzer in das Textfeld hineinschreiben darf.
```

Aufgabe 1:



a) Wenn die Maus den Button *Nein* berührt, verschwindet dieser und am rechten Rand taucht ein analoger, zweiter *Nein*-Button auf. Wenn man auf den *Ja*-Button klickt, erscheint ein Label mit der Aufschrift „*Schade!*“. Beachte auch den Titel dieses Formblattes!

b) Ändere dieses Programm so ab bzw. schreibe es neu, dass **nach** Betätigung des *Ja-Buttons* die Fragestellung in „Bist du schlau?“ geändert wird und als Antwort „schön!“ erscheint.

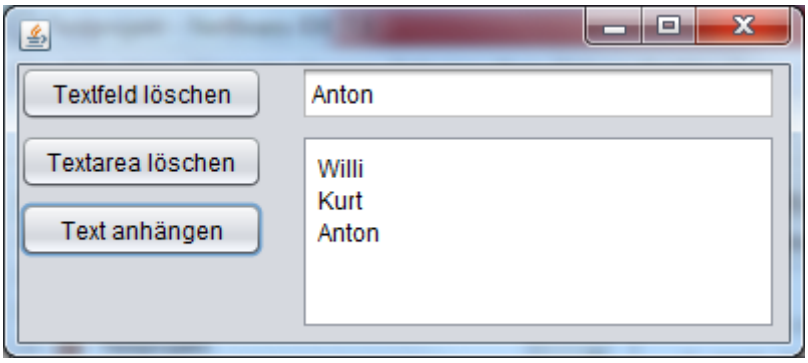
Text-Area

Eine ***Textarea*** besitzt ebenfalls viele Methoden und Eigenschaften. Interessant für uns wären hauptsächlich nur die folgenden Methoden:

```
public void setText(String  s)
        // Um alles zu löschen, gibt man einen Leerstring ("") ein.

public String getText()

public void append(String  s)
        // Ein Zeilensprung wird durch die Zeichenfolge  "\n"  erzwungen.
```

Aufgabe 2:

Erstelle ein Formblatt mit drei *Buttons*, einem *Textfield* und einer *Textarea*! Ein Mausklick auf den ersten *Button* löscht das Textfeld. Der zweite Button löscht den Inhalt der *Textarea*. Der dritte Button sorgt dafür, dass der Inhalt des Textfeldes (mit Zeilensprung) in die *Textarea* angehängt wird.



Lösung der Aufgabe 1a:

```
public class KlasseDumm extends javax.swing.JFrame {

public KlasseDumm() {
   initComponents();
   btNein2.setVisible(false);
   lbAussage.setVisible(false);
}

private void btnNein1MouseEntered(java.....){
   btNein1.setVisible(false);
   btNein2.setVisible(true);
}

private void btnNein2MouseEntered(java.....) {
   btNein2.setVisible(false);
   btNein1.setVisible(true);
}

private void btnJaMouseClicked(java.....) {
   btNein1.setVisible(false);
   btNein2.setVisible(false);
   btJa.setEnabled(false);
   lbAussage.setVisible(true);
}
} //Ende der Klasse KlasseDumm
```

Bemerkungen:
Zur Lösung der Aufgabe 1a wurde eine Klasse namens *KlasseDumm* erstellt. Von dieser Klasse wird natürlich nur ein einziges Objekt erzeugt, nämlich das, welches der Benutzer als sog. Formblatt (vom Typ *JFrameForm*) auf seinem Bildschirm sieht.
Dieses einzige Objekt besitzt einige offensichtliche Eigenschaften (Hintergrundfarbe, Größe, Abstand vom linken oder oberen Bildschirmrand, Sichtbarkeit usw.) und viele zusätzliche Methoden (es könnte z.B. seine Hintergrundfarbe ändern, wenn man auf das Formblatt klickt), die selbst professionellen Programmierern nicht alle bekannt sind. Außerdem besitzt es sog. *Komponenten*, das sind Objekte anderer Klassen (*Buttons*, *Label*, *Textfields* usw.).

Wenn das Programm gestartet wird, wird als erstes (natürlich!) der sog. *Konstruktor* der Klasse *KlasseDumm* aufgerufen und ein (bzw. hier **das**) Objekt wird konstruiert.

Der *Konstruktor* der Klasse *KlasseDumm* erzeugt das entsprechende Formblatt und erzeugt auch automatisch (mithilfe der Methode *initComponents*) alle Komponenten, die zu einem (**dem**) Objekt der Klasse *KlasseDumm* gehören.

Im Quelltext ist der Konstruktor daran zu erkennen, dass die entsprechende Konstruktormethode denselben Namen wie die Klasse hat. Außerdem besitzt diese Methode die Eigenschaft *public* (logisch!) und zwischen dem reservierten Wort *public* und dem Methodennamen (=Klassenname) steht kein Rückgabewert der Methode, weil der Rückgabewert natürlich klar ist: ein neues (bzw. das) Objekt der Klasse *KlasseDumm*.

Fast immer möchte der Programmierer, dass direkt nach dem Start bestimmte Anfangsbedingungen erfüllt sind (in der obigen Aufgabe 1a sollen u.a. ein *Button* und ein *Label* unsichtbar sein). Diese Anfangsbedingungen bzw. die entsprechenden Java-Anweisungen schreibt man grundsätzlich in den Klassenkonstruktor des Formblattes (natürlich **hinter** den Befehl *initComponents*, weil logischerweise z.B. ein Button erst erzeugt werden muss bevor man ihn unsichtbar machen kann).

Die sog. *Event*-Methoden wie z.B. **`btnJaMouseClicked(java.....)`** lassen sich im Quelltext nicht so einfach löschen (also nicht einfach markieren und anschließend mit der <*entf*>-*Taste* löschen). Sie können aber folgendermaßen gelöscht werden:
Markiere im Design-Fenster das entsprechende Objekt (also z.B. den Button) und wähle anschließend mit der rechten Maustaste die Option *properties*. Wähle im dann erscheinenden Fenster das Menü *Events*. Dort kann man das entsprechende *Event* auswählen und löschen.

Lösung der Aufgabe 2:

```
public class Aufgabe2 extends javax.swing.JFrame {

  public Aufgabe2() {
    initComponents();
    tf.requestFocus();
  }

  private void btTfLoeschenMouseClicked(java….) {
    tf.setText("");
    tf.requestFocus();
  }

  private void btTaLoeschenMouseClicked(java….) {
    ta.setText("");
  }

  private void btAnhaengenMouseClicked(java…..) {
    String wort = tf.getText();
    ta.append(wort);
    ta.append("\n");
  }
}
```
//Ende der Klasse Aufgabe2

Zahlentypen

`int` Ganze Zahl im Bereich von -2^{31} bis $+2^{31}-1$
Hinweis: $2^{31} \approx 2{,}1$ Milliarden $= 2{,}1 \cdot 10^9$

`long` Ganze Zahl im Bereich von -2^{63} bis $+2^{63}-1$
Hinweis: $2^{63} \approx 9{,}2 \cdot 10^{18}$

Ganze Zahlen werden rechnerintern als Binärzahlen (=Dualzahlen) dargestellt. Ein Beispiel für eine 8-bit-Dualzahl wäre etwa

0	**0**	**1**	**0**	**0**	**1**	**1**	**1**

VZ 7-stellige Dualzahl

Obige Darstellung entspricht der Dezimalzahl +39

Bei negativen ganzen Zahlen ist übrigens die interne Darstellung relativ kompliziert; nicht nur das Vorzeichen-Bit hat dann einen anderen Wert, auch der Betrag wird anders dargestellt.

Reelle Zahlen (sog. Gleitkommazahlen) werden rechnerintern folgendermaßen dargestellt:

Gleitkommazahl = (Vorzeichen) · Mantisse · 2^{Exponent}

0	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**

VZ Exponent Mantisse

Die exakte Bedeutung der einzelnen Bits (u.a. Vorzeichen in Exponent und Mantisse) kann je nach Programmiersprachenversion unterschiedlich sein. Um möglichst schnell Rechnungen durchführen zu können, wird nicht unbedingt die naheliegendste Interpretationsmöglichkeit obiger Darstellung genommen.

Die Größe (Anzahl der Bits) einer Zahl hängt vom Typ, vom Rechner und von der Programmiersprache ab.

Gleitkommazahlen sind z.B. 2.3456E−7 oder -12.3 oder auch 25
Das Komma in der Dezimalzahldarstellung wird in Java grundsätzlich durch den Punkt realisiert!

`double` Dezimalzahl im Bereich von
$-1{,}797693134862 \cdot 10^{308}$ bis $+1{,}797693134862 \cdot 10^{308}$
Die **betragsmäßig** kleinste, positive *double*-Zahl besitzt den Wert: 2.22510^{-307}
Allerdings kann dieser Datentyp nur mit einer Genauigkeit von 17 Stellen speichern. Die beiden Zahlen 0.000000012345678901234567**1** und 0.000000012345678901234567**2** sind also bei Rechnungen in Java für den Rechner identisch.

Die mathematische Division wird mithilfe von **`"/"`** vorgenommen. Wenn auf beiden Seiten dieses **Divisionszeichens** ein Wert vom Typ *int* steht, so ist auch das Ergebnis wieder vom Typ *int* und zwar der ganzzahlige Anteil der Division. Dies ist eine häufig vorkommende Fehlerquelle. Betrachte dazu das folgende Beispielprogramm:

```
.....
int i;
double x;
double y;
i = 19/7;
x = 19/7;
y = 19.0/7;
ta.append("i = " + i + "\n");
ta.append("x = " + x + "\n");
ta.append("y = " + y + "\n");
.....
```

Obiges Programm liefert folgende Ausgabe:

```
i = 2
x = 2.0
y = 2.7142857142857144
```

Bemerkung: die oben dargestellte letzte Ziffer von y ist mathematisch offensichtlich falsch gerundet, wird aber vom Java-System so ausgegeben. Es muss sich also um eine Ungenauigkeit bei der Berechnung von **`19.0 / 7`** an der 17ten Stelle handeln.

Wichtiger Hinweis:
Die Abfrage *if (1/5 + 4/5 == 1)* liefert den Wert *false*.
Die Abfrage *if (1.0/5 + 4.0/5 == 1)* liefert den Wert *true* (was allerdings auch nicht selbstverständlich ist, wie wir später sehen werden).
Die Abfrage *if (1.0/5 + 4/5 == 1)* liefert den Wert *false*.

Den ganzzahligen Rest bei einer Integerdivision erhält man mit dem Zeichen „%“ (diese Rechnung wird „modulo“ genannt)
Beispiel: **`int n = 19 % 7`** liefert **`n = 5`**

Den Absolutbetrag einer Zahl erhält man z.B. so: n = Math.abs(-17);
bzw. x = Math.abs(-23.72).

Type-Casting

Unter Type-Casting versteht man die Umwandlung von verschiedenen Datentypen ineinander, z. B. die Umwandlung einer Zahl in eine Zeichenkette.

Im Folgenden werden folgende Beispielsvariablen verwendet:

```
double d = 4.5;
int i = 69;
long l = 123456789;
String s = "709";
```

Eine Integerzahl wird bei Bedarf automatisch umgewandelt:
l = i bzw. d = i sind immer möglich. Aber die umgekehrten Zuweisungen i = l bzw. i = d sind aus einleuchtenden Gründen nicht möglich (l könnte größer sein als der Integerbereich; d könnte Nachkommastellen besitzen).

Andere Umwandlungen müssen explizit ausgeführt werden:
Das Umwandeln einer **Gleitkommazahl in eine Ganzzahl** läuft auf einen Rundungsvorgang hinaus. Dafür gibt es zwei Varianten:
a) Abschneiden der Nachkommastellen (aus 2.8 wird die Zahl 2):
b) Mathematisches Runden der Zahl (aus 2.8 wird die Zahl 3):
Betrachte dazu das folgende Beispielprogramm:

```
…….
int i, k;
i = (int) 2.8;
ta.append("i = " + i + "\n");
k = (int) Math.round(2.8);
ta.append("k = " + k + "\n");
i = (int) -7.8;
ta.append("i = " + i +"\n");
k = (int) Math.round(-7.8);
ta.append("k = " + k + "\n");
……..
```

Obiges Programm liefert folgende Ausgabe:

i = 2
k = 3
i = -7
k = -8

Hinweis: Java rundet bei negativen Zahlen die Ziffer 5 anders als mathematisch gewohnt. Dies machen angeblich viele Programmiersprachen so.
Beispiel:
`k = (int) Math.round(-7.5)` liefert das Ergebnis k = -7.

Erklärung: Beim Runden einer Zahl mit **`Math.round`** addiert Java intern zuerst die Zahl 0.5 und schneidet anschließend vom Ergebnis die Nachkommastellen ab.

Bemerkung: Das Ergebnis von **Math.round** ist immer entweder vom Typ *int* oder vom Typ *long*.

Wenn man Zahlen mithilfe der Methode *setText()* ausgeben möchte, wird als Parameter der Datentyp *String* verlangt. Dies wird automatisch mithilfe von „+“ durchgeführt – vorausgesetzt, die Operation enthält eine Zeichenkette, die auch leer sein darf:

```
String s;
int i = 17;
double d = 23.0537;
s = "" + i;  // Umwandlung int → String
s = "" + d;  // Umwandlung double → String
```

Wenn man Strings in Zahlen umwandeln möchte, so funktioniert dies folgendermaßen:

```
i = Integer.parseInt(s); // Umwandlung String → int
d = Double.parseDouble(s); // Umwandlung String → double
```

Für den Fall, dass die Zeichenkette nicht richtig umgewandelt werden kann, erfolgt ein Laufzeitfehler.

Aufgabe:
Der Benutzer gibt in zwei Textfeldern jeweils eine Zahl mit beliebig vielen Nachkommastellen ein. Nach Anklicken eines Buttons berechnet der Computer die Summe dieser beiden eingegeben Zahlen und gibt sie mit zwei Nachkommastellen in einem dritten Textfeld aus.

Lösungsweg: Multipliziere zunächst das Ergebnis mit 100, runde es auf eine ganze Zahl, dividiere durch 100 und gib das neue Ergebnis aus.
Bemerkung: Falls das Ergebnis zur Darstellung keine 2 Nachkommastellen benötigt (Beispiel: 24.48 / 8.16 = 3.0), so soll das auch akzeptiert werden.

Lösung der Aufgabe:

```
private void tbAddiereMouseClicked(java….)    {
  double z1 = Double.parseDouble(tf1.getText());
  double z2 = Double.parseDouble(tf2.getText());
  double ergebnis = z1 + z2;
  ergebnis = 100*ergebnis;
  ergebnis = Math.round(ergebnis);
  ergebnis = ergebnis/100;
  tfErgebnis.setText(""+ergebnis);
}
```

Die *if-else*-Anweisung gilt als ein einziger Befehl. Deshalb sind im folgenden Beispiel die geschweiften Klammern überflüssig:

```
if (n > 0)  {
   if (n < 7) tfAusgabe.setText(“das ist eine Note“);
   else tfAusgabe.setText(“zwar positiv, aber keine Note“);
}
```

Man kann obiges Beispiel also völlig gleichwertig auch so schreiben:

```
if (n > 0)
   if (n < 7) tfAusgabe.setText(“das ist eine Note“);
   else tfAusgabe.setText(“zwar positiv, aber keine Note“);
```

Bemerkung:
Die Anweisung *else* bezieht sich, falls keine geschweiften Klammern dies anders regeln, immer auf die direkt vorhergehende *if*-Abfrage.
Im obigen Beispiel wird der else-Teil also ausgeführt, wenn $n \geq 7$ ist.

Im folgenden Beispiel wird der else-Teil ausgeführt, wenn $n \leq 0$ ist:

```
if (n > 0)  {
   if (n < 7) tfAusgabe.setText(“das ist eine Note“);
}
else tfAusgabe.setText(“Noten müssen positiv sein!“);
```

Computerfehler bei Integer-Rechnungen

```
private void  btStartMouseClicked(java.awt……) {
  int x = 2 000 000 000;
  int y = 2 000 000 000;
  int z = x + y;
  z = z/1000;
  taAusgabe.append("" + z + "\n");
  taAusgabe.append("" + (x/1000 + y/1000));
}
```

Obiges Programm liefert im Ausgabebereich folgende zwei Zahlen untereinander:

```
-294 967
4 000 000
```

Erklärung: nur bei der ersten Rechnung wird der gültige Zahlenbereich für Integerzahlen überschritten. Mathematisch betrachtet, sollte beides mal 4 000 000 das Ergebnis sein.
Das bedeutet:
Das Distributivgesetz (a + b) / c = a/c + b/c gilt im Computerbereich nicht uneingeschränkt!

Betrachte folgendes Programm:

```
private void btStartMouseClicked(java.awt……..) {
  int x = 2 000 000 000;
  int z = 2*x;
  z = z/2;
  taAusgabe.append("" + z + "\n");
}
```

Dieses Programm liefert als Ausgabe die falsche Zahl **-147 483 648**
Selbst der Befehl int z = x*2/2 würde zu derselben falschen Ausgabe führen.

Das bedeutet:
Die Rechenregel a * b / b = a gilt im Computerbereich nicht uneingeschränkt!

In den obigen beiden Programmen ist der Fehler noch ziemlich schnell erkennbar, denn das mathematisch richtige Ergebnis kann nicht negativ sein. Wesentlich weniger auffällig ist das falsche Ergebnis, welches folgendes Programm liefert:

```
private void btStartMouseClicked(java.awt……) {
  int x = 2 000 000 000;
  int y = x + 1 598 402 136;
  int z = y/2 + 712 346 123;
  taAusgabe.append("" + z + "\n");
}
```

Dieses Programm liefert als Ausgabe die Zahl **`364 063 543`**
Das richtige Ergebnis wäre allerdings **`2 511 547 191`**

Zusammenfassend sollte man grundsätzlich Folgendes beachten:

Wenn bei Rechnungen mit Integer-Zahlen der gültige Zahlenbereich über- oder unterschritten wird, so entstehen mathematische Fehler, die vom Computersystem leider nicht als falsch erkannt werden.

Computerfehler bei Rechnungen mit Gleitkommazahlen

```
private void jButton1MouseClicked(java…..)    {
    double x = 10;
    for (int i=1; i<=100;i++)    x = x-0.1;
    x = x * 1E20;
    taAusgabe.append("" + x);
}
```

Bei exakter Rechnung müsste das obige Programm die Zahl 0 ausgeben.
Tatsächlich wird jedoch (in Java, 2013) 1 879 052,4691780775 ausgegeben.

Vorsicht auch bei direkten Vergleichen mit Gleitkommazahlen! Das folgende Programm sollte eigentlich nur fünfmal das Wort *Hallo* ausgeben. Stattdessen liefert es im Prinzip eine Endlosschleife, welche nur durch die aus Sicherheitsgründen zusätzlich angebrachte Bedingung **`&& (x <= 10)`** verhindert wird.

```
private void
btStartMouseClicked(java.awt.event.MouseEvent evt) {
   double x = 0;
   while ((x != 1) && (x <= 10))  {
      taAusgabe.append("Hallo\n");
      x = x + 1.0/5;
   }
}
```

Obiges Programm gibt leider 50 Mal untereinander das Wort „Hallo“ aus.

Erklärung:
Der Computer kann reelle Zahlen, welche sehr viele Nachkommastellen haben, nicht exakt darstellen, weil er für die Speicherung von reellen Zahlen nur eine begrenzte Anzahl von Speicherzellen zur Verfügung stellt. Im Dezimalsystem hat die Zahl $\frac{1}{5} = 0{,}2$ zwar nur eine Nachkommastelle, aber im Binärsystem (Basis = 2) besitzt diese Zahl unendlich viele Nachkommastellen. Deshalb kann der Bruch $\frac{1}{5}$ vom Computer, der mit Dualzahlen rechnet, nicht exakt dargestellt

werden, und die Summe $\frac{1}{5}+\frac{1}{5}+\frac{1}{5}+\frac{1}{5}+\frac{1}{5}$ ist für den Computer eben nicht gleich 1.
Bemerkung: Wie genau das Java-Programm rechnet, kann durchaus vom Betriebssystem des Computers und von der Java-Version abhängen. Obiges Programm bezieht sich auf die Versionsdaten des Jahres 2013. Bei entsprechender Vergrößerung der obigen *while-Schleife* wird der Fehler aber in jeder Version nachweisbar sein!

Man darf bei Rechnungen mit Gleitkommazahlen niemals auf Gleichheit abfragen!

Beispiel:
die Abfrage *if (5 * x == 1)* ist bei Computern verboten. Stattdessen sollte man etwa folgendermaßen programmieren:
```
if (Math.abs(5*x – 1) < 0.0000001))  {…..}
```
Diese Vorgehensweise ist natürlich auch problematisch.

Übungsaufgaben

Für alle Aufgaben gibt der Benutzer in ein oder mehrere *Text Fields* die benötigten Zahlen ein. Nach Anklicken eines Buttons wird das Ergebnis in einem weiteren *Text Field* ausgegeben!

INTEGER-Rechnungen

1. Die Kantenlänge eines Würfels wird eingegeben. Das Würfelvolumen wird ausgegeben.
2. Länge und Breite eines Rechteckes werden ein-, Flächeninhalt und Umfang werden ausgegeben.
3. Aus Länge, Breite und Höhe eines Quaders werden das Volumen und die Oberfläche berechnet.
4. Eine eingegebene Anzahl von Jahren soll in Stunden umgerechnet werden. Schaltjahre werden dabei nicht berücksichtigt. Ein Jahr soll 365 Tage haben.
5. Ein Liter Benzin kostet 152 Cent. Nach Eingabe der getankten Benzinmenge (ganzzahlige Anzahl von Litern) wird der Benzinpreis ausgegeben (möglichst in Euro und Cent, also z.B. „65,36 €“).
6. Eine in mm eingegebene Länge soll der Computer in m, dm, cm und mm angeben.
   Beispiel : 1345mm = 1m 3dm 4cm 5 mm

7. Der Geldautomat einer Sparkasse soll den eingegebenen Betrag (≤2000€) in möglichst großen Scheinen ausgeben. Die Scheine gibt es bekanntlich mit folgenden Beträgen: 200€, 100€, 50€, 20€, 10€ und 5€. Der eingegebene Betrag muss also durch 5 teilbar sein.
   Beispiel: Eingabewert = 225€, Ausgabescheine: 200€, 20€ und 5€.

Lösungen

Aufgabe 1

```
private void btStartMouseClicked(java…) {
  double laenge, volumen;
  String s;
  s = tfEingabe.getText();
  laenge = Double.parseDouble(s);

  volumen = laenge*laenge*laenge;
  tfAusgabe.setText(""+volumen);
}
```

Aufgabe 2

```
private void btStartMouseClicked(java…) {
  double laenge, breite, flaeche, umfang;
  String s = tfLaenge.getText();
  laenge = Double.parseDouble(s);

  s = tfBreite.getText();
  breite = Double.parseDouble(s);

  flaeche = laenge*breite;
  tfFlaeche.setText(""+flaeche);

  umfang = 2*(laenge + breite);
  tfUmfang.setText(umfang + "");
}
```

Aufgabe 3

```
private void btStartMouseClicked(java…) {
  double l, b, h, flaeche, volumen;

  String s = tfLaenge.getText();
  l = Double.parseDouble(s);

  s = tfBreite.getText();
  b = Double.parseDouble(s);

  s = tfHoehe.getText();
  h = Double.parseDouble(s);
```

```
  flaeche = 2*l*b + 2*l*h + 2*b*h;
  tfFlaeche.setText(""+flaeche);

  volumen = l*b*h;
  tfVolumen.setText(volumen + "");
}
```

Aufgabe 4

```
private void btStartMouseClicked(java…) {
  String s = tfEingabe.getText();
  int j = Integer.parseInt(s);

  int h = j*365*24;
  tfAusgabe.setText("" + h);
}
```

Aufgabe 5

```
private void btStartMouseClicked(java…) {
  String s = tfLiter.getText();
  double preis = 1.52 * Double.parseDouble(s);
  preis = (int) (100 * preis);
  preis = preis/100;
  tfPreis.setText(preis + "€");
}
```

Aufgabe 6

```
private void btStartMouseClicked(java…) {
  String s = tfMM.getText();
  zahl = Integer.parseInt(s);
  int mm = zahl%10;
  zahl = zahl/10;
  int cm = zahl%10;
  zahl = zahl/10;
  int dm = zahl%10;
  zahl = zahl/10;
  tfAusgabe.setText(zahl + "m " + dm + "dm " + cm +
                                    "cm " + mm + "mm");
}
```

Logische Operatoren

Mit logischen Operatoren werden Wahrheitswerte verknüpft. Java bietet die Operatoren *Und* (& bzw. &&), *Oder* (| bzw. ||), *Xor* (^) und *Nicht* (!) an. *Xor* ist eine Operation, die genau dann falsch zurückgibt, wenn entweder beide Operatoren wahr oder beide falsch sind. Sind sie unterschiedlich, so ist das Ergebnis wahr.

Eine Besonderheit sind die sog. *Kurzschlussoperatoren* && für *Und* beziehungsweise || für *Oder*. Bei Benutzung dieser Kurzschlussoperatoren (welche eigentlich hauptsächlich verwendet werden) wird eine logische Bedingung nur dann weiter ausgewertet, wenn sie das Schlussergebnis noch beeinflussen kann. Ansonsten optimiert der Compiler die Programme, indem er zum Beispiel bei zwei Bedingungen die gesamte Auswertung schon beendet, wenn nach Auswertung der ersten Bedingung das Gesamtergebnis klar ist.

Und: if ((x > 3) && (x < 10)) …… Ist schon die erste Bedingung falsch, so kann der Ausdruck nicht mehr wahr werden. Das Ergebnis ist falsch.

Oder: if ((x > 3) || (y > 17)) ……. Ist schon die erste Bedingung wahr, so ist auch der gesamte Ausdruck wahr.

Die beiden folgenden Abfragen sind leider nicht äquivalent:

Abfrage 1: if ((x < 5) || ($\frac{1}{x} < 5$)) …….

Abfrage 2: if (($\frac{1}{x} < 5$) || (x < 5)) …….

Falls x hier den Wert 0 haben sollte, so würde die Anweisung bei der ersten Abfrage ausgeführt werden, die zweite Abfrage würde allerdings zu einem Programmabsturz führen.

Es ist aber in einigen Fällen gewünscht, dass alle Teilabfragen ausgewertet werden, insbesondere wenn Methoden Nebenwirkungen haben, etwa Zustände ändern. Daher führt Java zusätzliche Nicht-Kurzschlussoperatoren | und & ein, die in einem komplexen Ausdruck alle Teilausdrücke auswerten.

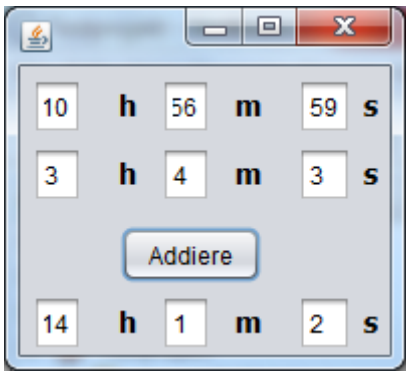
Für das ausschließende Oder *Xor* (Operator ^) kann es keinen Kurzschluss-operator geben, da immer beide Operanden ausgewertet werden müssen, bevor das Ergebnis feststeht.

Beispiel: In der folgenden zweiten und dritten Abfrage wird die Methode ***public boolean istPrimzahl(int n)*** aufgerufen:

```
if (2+3==5 ||  istPrimzahl(7))  ..... ;
if (2+3==5 &&  istPrimzahl(7))  ..... ;
if (2+3==4  &  istPrimzahl(7))  ..... ;
```

Integer-Aufgaben mit Bedingungen

1. Der Computer meldet, ob eine eingegebene Zahl a durch eine weitere eingegebene Zahl b teilbar ist. Verwende dazu den Modulo-Befehl!
2. Der Computer meldet, ob die drei eingegebenen, natürlichen Zahlen a, b und c ein pythagoräisches Tripel bilden. Löse die Aufgabe zuerst unter der Voraussetzung, dass c die größte der drei Zahlen ist, danach ohne diese Voraussetzung!
3. Der Rechner überprüft, ob eine eingegebene Zahl im Intervall [4; 10] liegt.
4. Nach Eingabe von drei Zahlen soll der Computer die kleinste Zahl ausgeben.
5. Der Computer soll "Uhrenadditionen" durchführen. Die Summe soll nicht größer als ein Tag sein.
6. Der Computer soll (ähnlich wie in Aufgabe 5) Zeitdifferenzen berechnen können. Alle Zeitangaben sollen kleiner als ein Tag sein. Die erste Zeiteingabe ist später als die zweite. Tipp: Berechne zuerst die gesamte Zeitdifferenz in Sekunden!



Lösungen

Aufgabe 1

```
private void btStartMouseClicked(java…..) {
  int a = Integer.parseInt(tf1.getText());
  int b = Integer.parseInt(tf2.getText());
  if (a % b == 0) tfAusgabe.setText(a +" ist durch "
                                    + b +" teilbar");
  else tfAusgabe.setText(a + " ist nicht durch " + b
                                       + " teilbar");
}
```

Aufgabe 2

```
private void btStartMouseClicked(java…..) {
  int a = Integer.parseInt(tf1.getText());
  int b = Integer.parseInt(tf2.getText());
  int c = Integer.parseInt(tf3.getText());
  if ((a*a + b*b == c*c) || (a*a + c*c == b*b) ||
                                  (b*b + c*c == a*a))
  tfAusgabe.setText("es ist ein pyth. Tripel");
  else tfAusgabe.setText("es ist kein pyth. Tripel");
}
```

Aufgabe 3

```
private void btStartMouseClicked(java…..) {
  int a = Integer.parseInt(tf1.getText());
  if ((a<=10) && (a >= 4))
          tfAusgabe.setText(a + " liegt in [4;10]");
  else tfAusgabe.setText(a+" liegt nicht in [4;10]");
}
```

Aufgabe 4

```
private void btStartMouseClicked(java…..) {
  int a = Integer.parseInt(tf1.getText());
  int b = Integer.parseInt(tf2.getText());
  int c = Integer.parseInt(tf3.getText());
  int min = a;
  if (b < min) min = b;
```

```
  if (c < min) min = c;
  tfAusgabe.setText("die kleinste Zahl ist " + min);
}
```

Aufgabe 5 Lösungsweg 1:

```
private void btAddiereMouseClicked(java….) {
  int h1 = Integer.parseInt(tfh1.getText());
  int h2 = Integer.parseInt(tfh2.getText());
  int m1 = Integer.parseInt(tfm1.getText());
  int m2 = Integer.parseInt(tfm2.getText());
  int s1 = Integer.parseInt(tfs1.getText());
  int s2 = Integer.parseInt(tfs2.getText());

  int s3 = (s1 + s2) % 60;
  m1 = m1 + (s1 + s2)/60;
  tfs3.setText(""+s3);

  int m3 = (m1 + m2) % 60;
  h1 = h1 + (m1 + m2)/60;
  tfm3.setText(""+m3);

  int h3 = (h1 + h2);
  tfh3.setText(""+h3);
}
```

Aufgabe 5 Lösungsweg 2:

```
private void btAddiereMouseClicked(java….) {
  int h1 = Integer.parseInt(tfh1.getText());
  int h2 = Integer.parseInt(tfh2.getText());
  int m1 = Integer.parseInt(tfm1.getText());
  int m2 = Integer.parseInt(tfm2.getText());
  int s1 = Integer.parseInt(tfs1.getText());
  int s2 = Integer.parseInt(tfs2.getText());

  int zahls = (h1+ h2)*3600 + (m1 + m2)*60 + s1 + s2;

  int s3 = zahls % 60;
  tfs3.setText(""+s3);
  int zahlm = zahls / 60;
```

```
  int m3 = zahlm % 60;
  tfm3.setText(""+m3);
  int h3 = zahlm / 60;

  tfh3.setText(""+h3);
}
```

Aufgabe 6

```
private void btSubtrahiereMouseClicked(java….) {
  int h1 = Integer.parseInt(tfh1.getText());
  int h2 = Integer.parseInt(tfh2.getText());
  int m1 = Integer.parseInt(tfm1.getText());
  int m2 = Integer.parseInt(tfm2.getText());
  int s1 = Integer.parseInt(tfs1.getText());
  int s2 = Integer.parseInt(tfs2.getText());

  int delta = h1*3600+m1*60+s1 - (h2*3600+m2*60+ s2);

  int s3 = delta % 60;
  tfs3.setText(""+s3);
  int deltam = delta / 60;

  int m3 = deltam % 60;
  tfm3.setText(""+m3);
  int deltah = deltam / 60;

  tfh3.setText(""+deltah);
}
```

FOR-Schleifen

Die Syntax einer *for*-Schleife lautet:
`for (int i = 1; i <= 10; i=i+1) {Anweisungen}`
Üblicherweise wird hier die Laufvariable **`i`** deklariert und ihr Anfangswert definiert; eine Endbedingung wird erstellt und die Schrittweite kann gewählt werden. Damit ist diese Variable nur innerhalb der *for*-Schleife gültig. Hinter der *for*-Schleife ist diese Variable nicht mehr benutzbar.

1. Gib deinen Namen n mal untereinander aus!
2. a) Gib alle ganzen Zahlen zwischen 100 und 120 aus !
b) Gib alle ganzen Zahlen zwischen 100 und 120 rückwärts aus!
3. Gib die Zahlen 000, 001, 002, …, 999 als dreiziffrige Zahlen aus!
4. Gib zur Zahl n die Anzahl ihrer Teiler aus!
5. Gib für alle Zahlen von 1 bis 1000 jeweils die Anzahl ihrer Teiler aus!
6. Gib die Fakultät der Zahl n aus!
7. Berechne die n-te Potenz der Zahl 2 !
8. Es sollen die ersten 20 Zweierpotenzen ausgegeben werden.
9. Berechne die Summe aller natürlichen Zahlen
a) von 1 bis 100 b) von 100 bis 200.
10. Die Fibonacci-Folge lautet: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Der Computer soll die ersten 50 Zahlenglieder berechnen und ausgeben.
11. Auf einem Schachbrett wird auf das erste Feld ein Cent, auf das zweite zwei Cent, auf das dritte 4 Cent, auf das vierte acht Cent usw. gelegt. Schreibe ein Programm, welches für jedes Feld den entsprechenden Cent-Betrag angibt und zusätzlich die Gesamtsumme berechnet!
12. Gib alle natürlichen, pythagoräischen Zahlentripel aus, wobei die ersten beiden Zahlen kleiner als 100 sein sollen.
13. Eine eingegebene, natürliche Zahl wird daraufhin untersucht, ob sie eine Primzahl ist. Schreibe dafür eine Methode *boolean istPrimzahl(int n)* !
14. Gib alle Primzahlen aus, die kleiner als 1 000 sind!
15. Ermittle, wie viele Primzahlen es unter 10 000 000 gibt! Hinweis: Aus Zeitgründen wähle zuerst eine wesentlich kleinere Obergrenze!
a) Wie lange benötigt dein Programm? Unsere Schulrechner benötigten dafür im Jahre 2018 etwa 7 Sekunden (bei gutem Algorithmus).
b) Untersuche den prozentualen Anteil der Primzahlen unter allen Zahlen!

c) Ist der Zusammenhang zwischen der Obergrenze und der benötigten Zeit linear, quadratisch oder noch größer?

Lösungen

```
1. private void btStartMouseClicked(java. ……..) {
       int n = 10;
       for (int i = 1; i <= n; i=i+1) {
         taAusgabe.append("Willi\n");
       }
   }

2.b) private void btStartMouseClicked(java. ……..) {
       for (int i = 120; i >= 100; i=i-1) {
         taAusgabe.append("" + i +"\n");
       }
     }

3. private void btStartMouseClicked(java. ……..) {
     for (int i = 1; i < 1000; i++) {
       if (i<10) taAusgabe.append("00");
       else if (i<100) taAusgabe.append("0");
       taAusgabe.append("" + i + "\n");
     }
   }

4. private void btStartMouseClicked(java. ……..) {
     int n = Integer.parseInt(tfEingabe.getText());
     int anzahl = 0;
     for (int i = 1; i <= n; i++) {
       if (n%i==0) anzahl++;
     }
     taAusgabe.append("Anzahl Teiler: " + anzahl);
   }
```

```
5. private void btStartMouseClicked(java. ……) {
    for (int a=1;  a<=1000; a++) {
      int anzahl = 0;
      for (int b=1; b<=a; b++) if (a%b==0) anzahl++;
      ta.append(a+": Anzahl der Teiler = "+anzahl+"\n");
    }
   }

6. private void btStartMouseClicked(java. ……..) {
     int n = Integer.parseInt(tfEingabe.getText());
     int fak = 1;
     for (int i = 1; i <= n; i++) {
       fak = fak * i;
     }
     taAusgabe.append("Fakultät =  "+ fak);
   }


7. private void btStartMouseClicked(java. ……..) {
     int n = Integer.parseInt(tfEingabe.getText());
     int pot = 1;
     for (int i = 1; i <= n; i++) {
       pot = pot * 2;
     }
     taAusgabe.append("Potenz =  "+ pot);
   }

8. private void btStartMouseClicked(java. ……) {
     int potenz = 1;
     for (int a=1;  a<=20; a++) {
        potenz = 2*potenz;
        ta.append(potenz+"\n");
     }
   }

9.b) private void btStartMouseClicked(java. ……..) {
      int sum = 0;
      for (int i = 100; i <= 120; i++) {
        sum = sum + i;
      }
      taAusgabe.append("Summe =  "+ sum);
    }
```

```
10. private void btStartMouseClicked(java. ……..) {
     int f0 = 0;
     int f1 = 1;
     taAusgabe.append("f0 = 0\n");
     taAusgabe.append("f1 = 1\n");
     int f2;
       for (int i = 2; i <= 50; i++) {
         f2 = f0 + f1;
         taAusgabe.append("f"+i +" = " + f2 + "\n");
         f0 = f1;
         f1 = f2;
       }
   }
```

```
11. private void btStartMouseClicked(java. ……..) {
    // Ab dem 64. Feld wird der Zahlenbereich von long überschritten.
      long potenz = 1;
      ta.append("1: 1\n");
      double summe = 1;
      for (int n=2;  n<=63; n++) {
          potenz = 2*potenz;
          ta.append(n+": "+potenz+"\n");
          summe = summe+potenz;
      }
      double x = potenz;
      x = 2*x;
      ta.append("64: "+x+"\n");
      summe = summe + x;
      ta.append("Gesamtsumme: "+summe);
   }
```

```
12. private void btStartMouseClicked(java. ……) {
      for (int a=1;  a<=100; a++)
      for (int b= a+1; b<=500;b++)  {
        int c = (int) Math.round(Math.sqrt(a*a+b*b));
        if (a*a + b*b == c*c)
          ta.append(a+", "+b+", "+c+"\n");
      }
   }
```

```
13. private void btStartMouseClicked(java. ……) {
      int zahl;
      zahl = Integer.parseInt(tfEingabe.getText());
      if (istPrimzahl(zahl))
         taAusgabe.setText("Primzahl");
      else taAusgabe.setText("keine Primzahl");
    }

    private boolean istPrimzahl(int n)  {
      int wurzel = (int) Math.round(Math.sqrt(n));
      boolean istPrim = true;
      for (int i=2; i <= wurzel; i++)
         if (n % i == 0) istPrim = false;
      return istPrim;
    }
```

Weitere Aufgaben

1. Im Rahmen einer Fahndung muss Kommissar Info mehrere Zeugen nach einem Autokennzeichen befragen. Die Buchstaben haben alle Befragten übereinstimmend benannt, nur an die Zahl des Kennzeichens kann sich keiner genau erinnern. Einer der Zeugen behauptet, die beiden ersten Ziffern der vierziffrigen KFZ-Nummer seien einander gleich. Ein anderer Zeuge sagt aus, dass er nur die beiden letzten Ziffern habe erkennen können, aber diese seien mit Sicherheit einander gleich gewesen. Eine dritte Zeugin (offensichtlich eine Zahlenakrobatin) meint, die vierziffrige Autonummer stelle eine Quadratzahl dar, aber sie wüsste nicht mehr, welche. Aus diesen drei Aussagen ist es Kommissar Info gelungen, die fehlende Autokennzeichennummer zu ermitteln. Wie lautet sie? Hinweis: Die Methode **`Math.sqrt(x)`** liefert die Quadratwurzel (vom Typ double) der Zahl x.

2. Man bestimme alle Paare natürlicher Zahlen (x, y), die das folgende Gleichungssystem lösen:

$$x \cdot (y + 41) = 2001$$

$$(2y + 3x) \cdot \frac{y}{2} = 2002$$

3. Das rechtwinklige Dreieck mit den Seiten 6, 8 und 10 hat die Eigenschaft, dass die Flächenmaßzahl gleich der Umfangsmaßzahl ist. Gibt es noch mehr solche rechtwinklige Dreiecke mit natürlichen Seitenzahlen a, b, $c \leq 30\,000$? Im Jahre 1984 gab es dazu einen Wettbewerb am Goethe-Gymnasium. Dabei ging es weniger um die richtige Lösung (welche schnell gefunden wurde) als vielmehr um die benötigte Zeit (damals noch für a, b, $c \leq 1\,000$). Die meisten Lösungen benötigten etwa 130 Sekunden, eine gute Lösung etwa 30s und die beste Lösung etwa 1s. Damals hatten wir Rechner mit etwa 1 MHz Taktfrequenz. Hinweis: eine (zeitmäßig) schlechte Lösung auf einem 1 GHz-Rechner braucht für a, b, $c \leq 30\,000$ etwa 30s.

4. Im Physikunterricht der Mittelstufe wird oft ein bestimmter Gleichungstyp behandelt. Beispiel:

$$\text{Linsenglei chung}: \quad \frac{1}{b} + \frac{1}{g} = \frac{1}{f}$$

$$\text{parallele Widerstände}: \quad \frac{1}{R_1} + \frac{1}{R_2} = \frac{1}{R_{ges}}$$

$$\text{serielle Federn}: \quad \frac{1}{D_1} + \frac{1}{D_2} = \frac{1}{D_{ges}}$$

Natürlich möchte der Lehrer seinen Schülern Beispielsaufgaben mit möglichst glatten Zahlen geben. Also: Gesucht sind alle möglichen natürlichen Zahlen a, b, c ≤ 100 und a ≠ b, welche die Gleichung

$\frac{1}{a} + \frac{1}{b} = \frac{1}{c}$ erfüllen.

Lösungen

Aufgabe 1

```
private void btStartMouseClicked(java……..) {
  int zahl;
  double wurzel, differenz;
  for (int a = 1; a <= 9; a++)
     for (int b = 0; b <= 9; b++)  {
        zahl = 1100*a + 11*b;
        wurzel = Math.sqrt(zahl);
        differenz = wurzel - (int) wurzel;
        if (differenz < 0.000001)
          ta.append(zahl + "\n");
     }
  ta.append("fertig");
}
```

Aufgabe 2

```
private void btStartMouseClicked(java……..) {
  for (int x = 1; x <= 2001; x++)
     for (int y = 1; y <= 2002; y++)
        if ((x*(y+41)==2001)&&((2*y+3*x)*y==4004))
           ta.append("x = "+x+" und y = "+y+"\n");
  ta.append("fertig");
}
```

Bemerkung: Man könnte durch leichte mathematische Überlegungen die Definitionsbereiche für x und y noch deutlich einschränken, um das Programm wesentlich schneller zu machen. Aber das Programm ist in dieser simplen Form schnell genug.

Aufgabe 3

Eine **zeitlich** schlechte Lösung wäre die folgende:

```
private void btStartMouseClicked(java……..) {
  int U;
  int F;
  double hyp;
  int c;
  for (int a = 1; a <= 30000; a++) {
    for (int b = a+1; b <= 30000; b++) {
      hyp = Math.sqrt(a*a+b*b);
      if (Math.abs(hyp - Math.round(hyp)) < 0.0001) {
        c = (int) Math.round(hyp);
        U = a+b+c;
        F = a*b / 2;
        if (U == F)
          taAusgabe.append(""+a+", "+ b+ ", " + c+"\n");
      }// of if
    } // of for b
  } // of for a
  taAusgabe.append("fertig");
}
```

Aufgabe 4

```
private void btStartMouseClicked(java…..) {
  double c;
  for (int a = 1; a <= 100; a++) {
    for (int b = a+1; b <= 100; b++) {
      c = 1.0/(1.0/a + 1.0/b); //wichtig: Zahlentyp double !
      if (Math.abs(c - Math.round(c)) < 0.0001)
          taAusgabe.append("" + a + ", " + b + ", " +
                                    Math.round(c)+"\n");
    }// of for b
  } // of for a
}
```

Rekursive Methoden

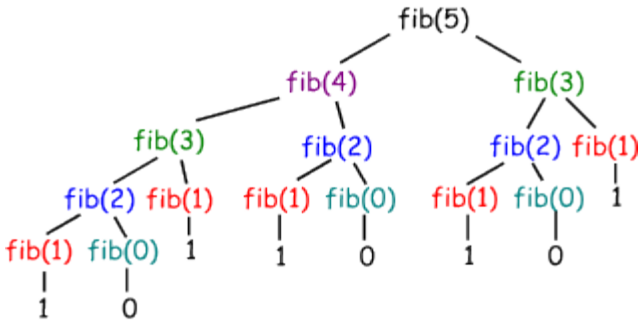
Rekursive Methoden sind Methoden, die sich selbst aufrufen. Im folgenden Beispiel werden die Fibonacci-Zahlen *rekursiv* ermittelt:

```
private int fib(int n)  {
   if (n == 0) return 0;
   else if (n == 1) return 1;
        else return (fib(n-1) + fib(n-2));
}
```

```
private void btStartMouseClicked(java……..) {
   for (int i=0; i <= 10; i++) {
      taAusgabe.append("f"+i + " = " + fib(i) + "\n");
   }
}
```

Es ist wichtig, dass der rekursive Aufruf nur unter einer Bedingung erfolgt, die irgendwann nicht mehr erfüllt wird. Wäre dies nicht der Fall, so würde sich die Methode unendlich oft selbst aufrufen und alle Ergebnisse der aufgerufenen Funktionen müssten irgendwo gespeichert werden: Das kann natürlich mangels Speicherplatz nicht funktionieren. Deshalb würde das Programm dann abstürzen.

Rekursive Methoden sind häufig leicht zu verstehen und sehr kurz. Andererseits sind sie grundsätzlich langsamer als die nicht-rekursiven Methoden (die *iterative* Methoden genannt werden). Obige rekursive Fibonacci-Methode ist besonders langsam. Das sieht man unmittelbar ein, wenn man grafisch darstellt, wie viele Methoden beim Aufruf von z.B. fib(5) berechnet werden müssen. Da ist die iterative Lösung, welche auf der Seite 32 dieses Skriptes steht, schon wesentlich schneller.



Hier kommt noch verlangsamend hinzu, dass Unterprogramme zur Berechnung einer bestimmten Fibonaccizahl mehrfach im Speicher vorhanden sind. Die Berechnung von Fibonaccizahlen ist also auf diese Weise nicht sehr effizient.

Aus diesem Grunde versucht man immer dann, wenn Geschwindigkeit wichtig ist, rekursive Programme durch iterative zu ersetzen. Ohne Beweis gilt:
Für jedes rekursive Programm gibt es eine äquivalente iterative Lösung.
Allerdings können diese iterativen Lösungen manchmal recht umständlich und kompliziert sein.

Interessante Bemerkung: Selbst bei der auf der Seite 32 dargestellten iterativen Berechnung der 50. Fibonacci-Zahl muss man immer noch die ersten 49 Fibonacci-Zahlen (mit einer Schleife) berechnen. Die Krönung der Geschwindigkeit wäre natürlich, wenn man die n-te Fibonacci-Zahl berechnen könnte, ohne vorher alle vorherigen berechnen zu müssen. Das ist ein mathematisches Problem, welches sich auf Schulniveau (Mathe-Leistungskurs) sogar lösen lässt: Für die n-te Fibonacci-Zahl gilt nämlich (ohne Beweis):

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Im folgenden Beispiel wird die Fakultät einer Zahl rekursiv berechnet.
Bekanntlich gilt: $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n = (n-1)! \cdot n$

```
private int fak(int n)  {
   if ((n == 1) || (n==0)) return 1;
   else return (fak(n-1) * n);
}


private void jButton1MouseClicked(java……)   {
    for (int i=1; i <= 10;i++) {
       taAusgabe.append(i + "! = " + fak(i) + "\n");
    }
}
```

Das folgende Beispiel zeigt einen rekursiven Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen. Er wird *Euklidischer Algorithmus* genannt. Er basiert auf der Idee, dass der gemeinsame Teiler für a und b der gleiche ist, wie der von b und r, wobei r der Rest bei der Division von a durch b ist: ggT(a, b) = ggT(b, r) mit r = a modulo b.

Beispielsrechnung: ggT(70, 42)
70 : 42 = 1 Rest 28
42 : 28 = 1 Rest 14
28 : 14 = 2 Rest 0
14 : 0

Im letzten Fall (Divisor = 0) ist der Dividend das gesuchte Ergebnis.
Der rekursive Algorithmus dazu lautet:

```
public int ggT(int a, int b)  {
   if (b == 0) return a;
   else return ggT( b, a % b );
}


private void jButton1MouseClicked(java…….) {
   taAusgabe.append(ggT(70, 42) +"\n");
}
```

Die drei letzten Beispiele waren insofern Spezialfälle, weil die rekursiven Aufrufe hier immer die letzten Befehle in der Methode waren. Oft stehen rekursive Aufrufe aber auch mitten innerhalb einer Methode. Wenn dann dieser rekursive Aufruf durchgeführt worden ist, werden danach anschließend die restlichen Befehle der Methode ausgeführt. Das ist durchaus gewöhnungs-bedürftig

Ermittle mit Papier und Bleistift, was die folgenden Programme bewirken werden. Bedenke, dass bei jedem Methodenaufruf der übergebene Parameter einen anderen Wert hat. Bedenke auch, dass eine Änderung dieses Übergabe-parameters innerhalb einer Methode hier keinen Einfluss auf die übergeordnete Methode hat!

Aufgabe 1

```
private void wasSollsWohlSein(int n)  {
   taAusgabe.append("" + n + "\n");
   if (n > 1) wasSollsWohlSein(n-1);
   taAusgabe.append("" + n*n + "\n");
}


private void btStartMouseClicked(java……)     {
   wasSollsWohlSein(3);
}
```

Aufgabe 2

```
public int foo(int p, int q )  {
   if (p == 1) return q;
   else return q*foo(p-1, q);
}


private void btStartMouseClicked(java…..)    {
   taAusgabe.append("" + foo(4, 3));
}
```

Aufgabe 3

```
public int foo(int p) {
   if (p == 1) return 1;
   else return p + foo(p-1);
}


private void btStartMouseClicked(java……)    {
   taAusgabe.append("" + foo(5));
}
```

Schreibe für folgende Aufgaben jeweils ein **rekursives** Programm.

1. Aus dem Textfeld *tfEingabe* wird eine kleine natürliche Zahl eingelesen und in der Textarea *taAusgabe* wird eine entsprechende Anzahl von * ausgegeben.
Beispiel: Eingabe = 8, Ausgabe = ********

2. Aus dem Textfeld *tfEingabe* wird eine Zahl vom Typ *int* eingelesen und im Textfeld *tfAusgabe* wird die Anzahl der Ziffern dieser Zahl ausgegeben.
Schreibe die Methode *int getAnzahlZiffern(int zahl)*;

3. Aus dem Textfeld *tfEingabe* wird eine Zahl vom Typ *int* eingelesen und in der Textarea *taAusgabe* werden die einzelnen Ziffern in umgekehrter Reihenfolge ausgegeben.

4. Aus dem Textfeld *tfEingabe* wird eine Zahl vom Typ *int* eingelesen und im Textfeld *tfAusgabe* wird die Quersumme ausgegeben.

5. Es soll die Potenz a^n berechnet werden, wobei a eine beliebige reelle Zahl und n eine natürliche Zahl sein soll. Die Basis wird aus dem Textfeld *tfBasis* eingelesen, der Exponent aus dem Textfeld *tfExponent*. Schreibe die rekursive Methode namens *double potenz(double a, int n)*

6. Der Benutzer wird aufgefordert, eine Zahl größer gleich 2 einzugeben. Anschließend wird eine entsprechend große Pfeilspitze der folgenden Form in einer Textarea ausgegeben:
Beispiel: Zahl = 2:

```
\
  \
  /
/
```

Beispiel: Zahl = 4:

```
\
  \
    \
      \
      /
    /
  /
/
```

Lösungen

Aufgabe 1

```
private void rekursiveAusgabe(int zahl)  {
  if (zahl > 0)  {
     taAusgabe.append("*");
     rekursiveAusgabe(zahl - 1);
  }
}


private void btStartMouseClicked(java……) {
  int zahl = Integer.parseInt(tfEingabe.getText());
  taAusgabe.setText("");
  rekursiveAusgabe(zahl);
}
```

Aufgabe 2

```
private  int getAnzahlZiffern(int n)  {
  if (n == 0) return 1;
  else if (n/10 == 0) return 1;
       else return 1 + getAnzahlZiffern(n/10);
}

private void btStartMouseClicked(java……) {
  int zahl = Integer.parseInt(tfEingabe.getText());
  tfAusgabe.setText("" + getAnzahlZiffern(zahl));
}
```

Aufgabe 3

```
private void rekursiveAusgabe(int zahl)  {
  int ziffer = zahl % 10;
  taAusgabe.append("" + ziffer);
  if (zahl/10 != 0) rekursiveAusgabe(zahl/10);
}
```

```
private void btStartMouseClicked(java……) {
  int zahl = Integer.parseInt(tfEingabe.getText());
  taAusgabe.setText("");
  rekursiveAusgabe(zahl);
}
```

Hinweis: Die beiden Variablen namens *zahl* in den Methoden *btStartMouseClicked()* und *rekursiveAusgabe()* besitzen nur denselben Namen; sie belegen aber unterschiedliche Speicherplätze und können daher auch unterschiedliche Werte besitzen. Außerdem haben sie unterschiedliche Gültigkeitsbereiche. Das Gleiche gilt auch in der nächsten Aufgabe.

Aufgabe 4

```
private int quersumme(int zahl)  {
  int ziffer = zahl % 10;
  if (zahl/10 == 0) return ziffer;
  else return ziffer + quersumme(zahl/10);
}

private void btStartMouseClicked(java……) {
  int zahl = Integer.parseInt(tfEingabe.getText());
  tfAusgabe.setText("" + quersumme(zahl));
}
```

Aufgabe 5

```
private  double potenz(double a, int n)  {
  if (a == 0) return 0;
  else if (n == 0) return 1;
       else return a* potenz(a, n-1);
}

private void btStartMouseClicked(java……) {
  double basis = Double.parseDouble(tfBasis.getText());
  int exponent = Integer.parseInt(tfExponent.getText());
  tfAusgabe.setText("" + potenz(basis, exponent));
}
```

Aufgabe 6

```
int zahl;   // globale Variable

private void btStartMouseClicked(java….) {
   zahl = Integer.parseInt(tfEingabe.getText());
   taAusgabe.setText("");
   zeichne(1);
}

private void zeichne(int p)  {
   for (int i=1; i <= p; i++) taAusgabe.append(" ");
   taAusgabe.append("\\\n");
   if (p<zahl) zeichne(p+1);
   for (int i=1; i <= p; i++) taAusgabe.append(" ");
   taAusgabe.append("/\n");
}
```

Der Datentyp char

Alle Zeichen des Typs *char* werden in Java mit 2 Byte gespeichert. Sie sind auf der Basis des *Unicode-Zeichensatzes* nummeriert. Die ersten 128 Zeichen entsprechen dem sog. *ASCII-Code* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange):

Code	**…0**	**…1**	**…2**	**…3**	**…4**	**…5**	**…6**	**…7**	**…8**	**…9**	**…A**	**…B**	**…C**	**…D**	**…E**	**…F**
0…	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1…	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2…	*SP*	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3…	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4…	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5…	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6…	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7…	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*DEL*

Die ersten 32 ASCII-Zeichencodes (von $00 bis $1F) sind für Steuerzeichen reserviert. Das sind Zeichen, die keine Schriftzeichen darstellen, sondern die zur Steuerung von Geräten dienen, die den ASCII verwenden (etwa Drucker). Steuerzeichen sind beispielsweise die Nummer $D für den *Wagenrücklauf* (**C**arriage **R**eturn) oder die Nummer $A für den *Zeilenvorschub* (**L**ine**F**eed) oder die Nummer $7 für einen akustischen Piepton (**Bel**l).

Code $20 ist das Leerzeichen (engl. *space* oder *blank*), das in einem Text als Leer- und Trennzeichen zwischen Wörtern verwendet und auf der Tastatur durch die Leertaste erzeugt wird.

Die Codes $21 bis $7E sind alle *druckbaren* Zeichen, die sowohl Buchstaben, Ziffern und Satzzeichen (siehe Tabelle) enthalten.

Die Ziffern 0 bis 9 haben also die dezimalen Codes 48 bis 57, der Großbuchstabe „A“ hat den Dezimalcode 65 usw. Die Codezahlen für Kleinbuchstaben sind um 32 größer als die Codes der entsprechenden Großbuchstaben.

Den Dezimalcode eines Char-Zeichens erhält man beispielsweise (mithilfe der Typumwandlung) **int n = (int) 'A';**

Das zum Zahlencode 66 gehörige Zeichen erhält man demnach (mithilfe der Typumwandlung) **char ch = (char) 66;**

Bei der Ausgabe von Texten haben einige Zeichen eine besondere Bedeutung, z.B. die Anführungszeichen **"** oder der Backslash **** , welcher sog. *Escape-Sequenzen* einleitet (wie etwa **\n** für einen Zeilensprung). Derartige Zeichen werden durch Voraussetzen eines Backslash-Zeichens *maskiert*, verlieren damit also ihre eigentliche Bedeutung.
Beispiel: **taAusgabe.append("Er sagte: \"Guten Tag!\"");**

Am Ende stehen zwei Anführungszeichen. Das erste ist mit dem Backslash maskiert, wird also ausgegeben. Das zweite ist das Ende des Strings. Die Ausgabe in der Textarea sieht dann so aus: *Er sagte: "Guten Tag!"*

Möchte man einen Backslash ausgeben, so funktioniert das so:
taAusgabe.append("\\");
Alternativ könnte man auch Folgendes schreiben:
```
taAusgabe.append((char) 92);
```

Der Typ *char* kann auch für for-Schleifen benutzt werden:
```
for (char ch = 'A'; ch <= 'Z'; ch++) taAusgabe.append(""+ch);
```

Möchte man aus einem Textfeld ein *char*-Zeichen auslesen, so geht das so:
```
  char x = tfEingabe.getText().charAt(0);
```
Damit wird das 0-te Zeichen des entsprechenden Strings ausgelesen.

Der Typ *char* läßt sich auch in switch-case Abfragen einsetzen:

```
char ch = tfEingabe.getText().charAt(0);
switch (ch)   {
  case 'A':tfAusgabe.setText("hAllo"); break;
  case 66: tfAusgabe.setText("Bello"); break;
  // 66 ist die Codezahl des Buchstabens  'B'
  default:tfAusgabe.setText("Vorgabe");
}
```

Bei *switch*-Anweisungen werden immer alle nachstehenden Anweisungen bis zum nächsten *break* ausgeführt!

```
……
case  5: Anweisung 1;
         ……..;
         Anweisung N;
         break;
default: Anweisung 1;
         ……..;
         Anweisung M;
         break;
}
```

Falls der Variablenwert keinem der obigen Werte entspricht, werden die *default*-Anweisungen bis zum *break* (oder bis zur schließenden Klammer der *switch*—Anweisungen) ausgeführt. Die *default*-Anweisung sollte immer am Ende stehen.

Das *break* kann auch ausgelassen werden, wenn es der letzte Fall ist.

Aufgaben

1. Schreibe eine Funktion **`char  nachfolger(char ch, int n)`**, welche den n-ten Nachfolger des Buchstabens ch im Alphabet liefert.

Beispiel: Die Anweisung **`char zeichen = nachfolger('A', 3)`** bewirkt, dass zeichen = `'D'`

Schreibe dazu ein Testprogramm, in dem man im Textfeld *tfEingabe* einen Buchstaben eingibt und der entsprechende Nachfolger im Textfeld *tfAusgabe* ausgegeben wird!

Hinweis: Der 3. Nachfolger von `'Z'` soll der Buchstabe `'C'` sein. Die Nummer des 3. Nachfolgers des n-ten Buchstabens im Alphabet wäre (n + 3) Modulo 26.

Die Aufgabe hat mehrere Schwierigkeiten: Sorge zunächst dafür, dass das Programm nur bei Großbuchstaben den 3. Nachfolger erstellt. Danach sorge dafür, dass es mit dem **n-ten** Nachfolger (nur) bei Großbuchstaben klappt. Danach verbessere dein Programm so, dass auch der n-te Nachfolger für Kleinbuchstaben ermittelt wird.

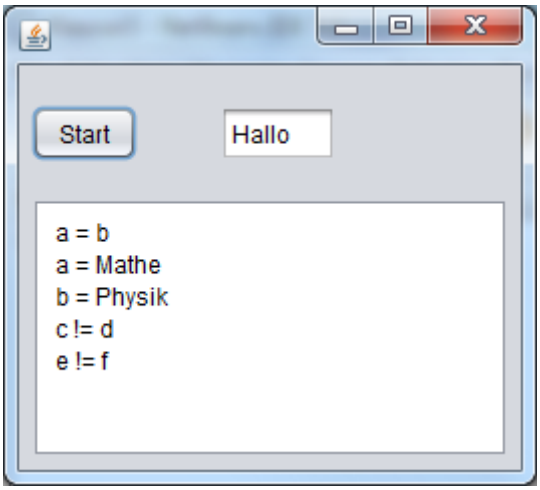
Lösung

Aufgabe 1

```
char  nachfolger(char ch, int n)  {
  int chNummer = (int) ch;
  int neueNummer = chNummer;
  if ((65 <= chNummer) && (91 >= chNummer ))
    neueNummer =  (chNummer - 65 +3)%26 + 65;
  else  if ((97 <= chNummer) && (123 >= chNummer ))
          neueNummer =  (chNummer - 97 +3)%26 + 97;
  ch = (char) neueNummer;
  return ch;
}

private void btStartMouseClicked(java……) {
  String s = tfEingabe1.getText();
  char ch = s.charAt(0);
  String t = tfEingabe2.getText();
  int anzahl = Integer.parseInt(t);
  tfAusgabe.setText(""+nachfolger(ch, anzahl));
}
```

Die Klasse String

Der Datentyp String nimmt in Java eine Sonderstellung ein. Er wird teils wie ein primitiver Datentyp und teils wie eine Klasse behandelt. Die Programmierung mit Strings ist in Java deshalb nicht so eindeutig.



Der Vergleich zwischen zwei Strings kann mit dem Operator == äußerst problematisch werden, wie das folgende Beispiel zeigt:

```
private void btStartMouseClicked(java.....)    {
    String a = "Mathe";
    String b = "Mathe";
    if (a == b) taAusgabe.append(" a = b   \n");
       else taAusgabe.append(" a != b   \n");

    tfEingabe.setText("Physik");
    b = tfEingabe.getText();
    taAusgabe.append(" a = " + a + "\n");
    taAusgabe.append(" b = " + b + "\n");

    String c = new String("Info");
    String d = new String ("Info");
    if (c == d) taAusgabe.append(" c = d   \n");
       else taAusgabe.append(" c != d   \n");

    tfEingabe.setText("Hallo");;
    String e = tfEingabe.getText();
    String f = tfEingabe.getText();
    if (e == f) taAusgabe.append(" e = f   \n");
       else taAusgabe.append(" e != f   \n");
}
```

Erklärung:
Man muss unterscheiden zwischen dem Namen für eine Variable und dem Inhalt einer Variablen. Der Name einer Variablen ist nur ein Synonym für die Nummer der ersten von mehreren hintereinander folgenden Speicherzellen, welche den zugehörigen Inhalt enthalten.
Der Operator `==` vergleicht nur die Variablennamen, also die Nummern der Speicherzellen miteinander.

Es ist klar, dass im obigen Programm die beiden Variablen c und d zwei unterschiedlichen Speicherstellen entsprechen. Diese Variablen c und d werden erst während der Laufzeit des Programms erzeugt.

Die beiden Variablen a und b werden jedoch schon vom Compiler erzeugt. Dieser weist der Variablen a den Speicherplatz zu, an dem die Konstante „Mathe“ steht. Der Compiler erkennt auch, dass die Variable b denselben konstanten Inhalt erhalten soll, und er weist der Variablen b dann auch denselben Speicherplatz wie der Variablen a zu. Trotzdem kann das Java-System später während der Laufzeit des Programms den beiden Variablen a und b immer noch unterschiedliche Werte zuweisen.

Bei der Speicherplatzzuweisung für die Variablen e und f kann der Compiler noch nicht wissen, welche Werte e und f später erhalten sollen.

Um obige Problematik zu umgehen, sollte man grundsätzlich bei inhaltlichen Stringvergleichen die Methode *equals* benutzen:
`if (a.equals(b))` …

Der +-Operator kann nicht nur mit numerischen Operanden verwendet werden, sondern auch zur Verkettung von Strings. Ist wenigstens einer der beiden Operatoren in **a + b** ein String, so wird der gesamte Ausdruck als String-Verkettung ausgeführt. Hierzu wird gegebenenfalls zunächst der Nicht-String-Operand in einen String umgewandelt und anschließend mit dem anderen Operanden verkettet. Das Ergebnis der Operation ist wieder ein String, in dem beide Operanden hintereinander stehen.
Beispiel: `String s = "Hans" + "wurst";`

Etwas Vorsicht ist geboten, wenn sowohl String-Verkettung als auch Addition in einem Ausdruck verwendet werden sollen, da die in diesem Fall geltenden Vorrang- und Assoziativitätsregeln zu unerwarteten Ergebnissen führen können. Die folgende Anweisung **`tfAusgabe.setText("3 + 4 = " +3 + 4)`** gibt daher nicht **3 + 4 = 7**, sondern **3 + 4 = 34** aus.

Um das gewünschte Ergebnis zu erzielen, müsste der Teilausdruck **3 + 4** geklammert werden: **`tfAusgabe.setText("3 + 4 = " +(3 + 4))`**

STRING-Methoden

`boolean  equals(Object anObject)`
Diese Methode liefert *true*, wenn das aktuelle String-Objekt und *anObject* gleichen Inhalt besitzen.
Beispiel:
```
String s = "7";
int n = 3 + 4;
if (s.equals("" + n)) ..... // liefert den Wert true
```

`boolean  equalsIgnoreCase(String s)`
Diese Methode ignoriert die eventuell vorhandenen Unterschiede in der Groß-/Kleinschreibung beider Zeichenketten. Das Vergleichsobjekt muss allerdings auch vom Typ *String* sein.
Beispiel:
```
String s = "goeTHe";
String t = "GOethe";
if (s.equalsIgnoreCase(t))  // liefert den Wert true
```

`int  indexOf(char ch)`
Diese Methode sucht das erste Vorkommen des Zeichens ch innerhalb des aktuellen String-Objekts. Wird ch gefunden, liefert die Methode den Index des ersten übereinstimmenden Zeichens zurück, andernfalls wird -1 zurück-gegeben. Dabei hat das erste Element eines Strings den Index 0 und das letzte den Index length() – 1 .
Beispiel:
```
String s = "Goethe";
int n = s.indexOf('e');      // liefert den Wert n=2
```

`int indexOf(String s)`

Diese Methode sucht das erste Vorkommen der Zeichenkette s innerhalb des aktuellen String-Objekts. Wird s gefunden, liefert die Methode den Index des ersten übereinstimmenden Zeichens zurück, andernfalls wird -1 zurück-gegeben. Dabei hat das erste Element eines Strings den Index 0 und das letzte den Index length() – 1 .

Beispiel:
```
String s = "Trallallalla";
int n = s.indexOf("all");      // liefert den Wert n=2
```

`String toLowerCase()`

Diese Methode liefert den String zurück, der entsteht, wenn alle Zeichen des aktuellen Objektes in Kleinbuchstaben umgewandelt werden. Besitzt das aktuelle Objekt keine umwandelbaren Zeichen, wird das aktuelle Objekt zurückgegeben.

Beispiel:
```
String s = "GOEThe12345+-*:";
s = s.toLowerCase(); // liefert s = "goethe12345+-*:"
```

`String toUpperCase()`

Diese Methode liefert den String zurück, der entsteht, wenn alle Zeichen des aktuellen Objektes in Großbuchstaben umgewandelt werden. Besitzt das aktuelle Objekt keine umwandelbaren Zeichen, wird das aktuelle Objekt zurückgegeben.

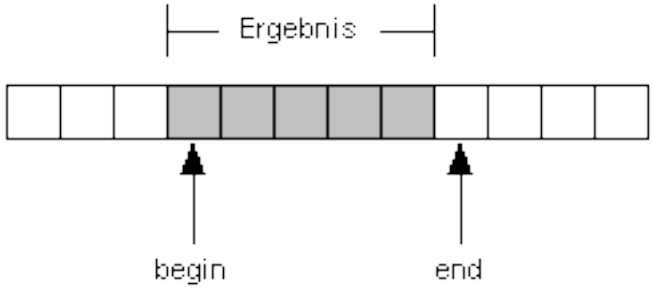
Beispiel:
```
String s = "goeTHE12345+-*:";
s = s.toUpperCase(); // liefert s = "GOETHE12345+-*:"
```

`char charAt(int index)`

Liefert das Zeichen an der Position index des aktuellen Stringobjektes. Dabei hat das erste Element eines Strings den Index 0 und das letzte den Index length() – 1 .

Beispiel:
```
String s = "Goethe";
char ch = s.charAt(3)  // liefert ch = 't'
```

`String substring(int begin, int end)`
Liefert den Teilstring, der an Position **begin** beginnt und an Position **end** endet. Wie bei allen Zugriffen über einen numerischen Index beginnt auch hier die Zählung bei 0.
Ungewöhnlich bei der Verwendung dieser Methode ist die Tatsache, dass der Parameter **end** auf das erste Zeichen hinter den zu extrahierenden Teilstring verweist (siehe Abbildung). Der Rückgabewert ist also die Zeichenkette, die von Indexposition **begin** bis Indexposition **end – 1** reicht.



Beispiel:
```
String s = "Goethe-Gymnasium";
String teil = s.substring(2,6); // liefert teil = 'ethe'
```

Bemerkung: die Anweisung **`String s = s.substring(3, 3)`** liefert einen Leerstring.

`String substring(int begin)`
Diese zweite Variante der Methode *substring* wird nur mit einem einzigen Parameter aufgerufen. Sie liefert den Teilstring von der angegebenen Position bis zum Ende des aktuellen Stringobjektes.
Beispiel:
```
String s = "Goethe-Gymnasium";
String teil = s.substring(4);
// liefert teil = 'he-Gymnasium'
```

`int length()`
Liefert die aktuelle Länge des aktuellen String-Objekts. Ist der Rückgabewert 0, so bedeutet dies, dass der String leer ist. Wird ein Wert *n* größer 0 zurückgegeben, so enthält der String *n* Zeichen, die an den Indexpositionen 0 bis *n* - 1 liegen.
Beispiel:
```
String s = "Goethe";
int n = s.length(); // liefert n = 6
```

Weitere Methoden im Zusammenhang mit Strings

Leider gibt es keine vordefinierte Methode für Strings, welche das n-te Zeichen innerhalb eines Strings löscht. Dafür muss man sich etwa folgende Methode schreiben:

```
private String deleteCharAt(String s, int n)    {
   String hilf;
   hilf = s.substring(0,n) + s.substring(n+1);
   return hilf;
 }
```

Durch die Anweisung **`wort = deleteCharAt(wort, 5)`** wird damit das fünfte Zeichen (beachte: die Zählung beginnt bei Null!) in der Variablen *wort* gelöscht.

Wenn man einen String s in eine Zahl umwandeln möchte (das geschieht sehr häufig bei einer Benutzereingabe, welche üblicherweise als String eingelesen wird), so funktioniert dies folgendermaßen:

```
i = Integer.parseInt(s);
d = Double.parseDouble(s);
```

Für den Fall, dass die Zeichenkette nicht richtig umgewandelt werden kann, erfolgt ein Laufzeitfehler.

STRING-Aufgaben

1. Ein Wort wird in das Textfeld *tfEingabe* eingegeben. Der Rechner gibt die Länge des Wortes im Textfeld *tfAusgabe* aus.

2. Ein Passwort soll abgefragt werden. Es wird in das Textfeld *tfEingabe* eingegeben. Das Passwort lautet "Goethe-Gymnasium". Das eingegebene Wort soll unabhängig von der Groß- und Kleinschreibung untersucht werden. Als z.B. soll ‚goEthe-gYmNasium' auch akzeptiert werden. Im Textfeld *tfAusgabe* wird entweder „akzeptiert" oder „nicht akzeptiert" ausgegeben.

3. Ein Text wird in das Textfeld *tfEingabe* eingegeben. Der Rechner gibt im Textfeld *tfAusgabe* aus, wie oft der Buchstabe ‚e' in diesem Text vorkommt. Bemerkung: die relative Häufigkeit der einzelnen Buchstaben hängt von der Sprache (deutsch, englisch usw.) ab. Man kann also bei einem längeren Text die Sprache aufgrund der Buchstabenhäufigkeit identifizieren.

4. Schreibe eine Funktion **String loescheZeichen(String s, int n)** , welche das n-te Zeichen (beachte: Nummerierung beginnt bei 0) im übergebenen String s löscht und den um ein Zeichen kleineren String zurückgibt. Dabei wird vorausgesetzt, dass der Parameter n kleiner ist als die Länge des Strings s.

5. Entferne aus einem im Textfeld *tfEingabe* eingegebenen Text alle Leerzeichen (CHR(32)) und gib ihn ohne Leerzeichen im Textfeld *tfAusgabe* wieder aus!

6. Schreibe eine Funktion **String fuegeZeichenEin(String s, int n, char ch)** , welche an der n-ten Stelle (beachte: Nummerierung beginnt bei 0) im übergebenen String s das Zeichen ch einfügt und den um ein Zeichen längeren String zurückgibt.. Dabei wird vorausgesetzt, dass der Parameter n kleiner ist als die Länge des Strings s.

7. Schreibe eine Funktion **String ersetzeZeichenDurch(String s, int n, char ch)**, welche an der n-ten Stelle (beachte: Nummerierung beginnt bei 0) im übergebenen String s das dort vorhandene Zeichen durch das Zeichen ch ersetzt und den neuen, gleich langen String zurückgibt. Dabei wird vorausgesetzt, dass der Parameter n kleiner ist als die Länge des Strings s.

8. Schreibe eine Funktion
String ersetzeZeichenDurch(String s, char chAlt, char chNeu),
welche alle im String vorhandenen Zeichen *chAlt* durch das neue Zeichen *chNeu* ersetzt. Dabei wird vorausgesetzt, dass die beiden Zeichen *chAlt* und *chNeu* ungleich sind.

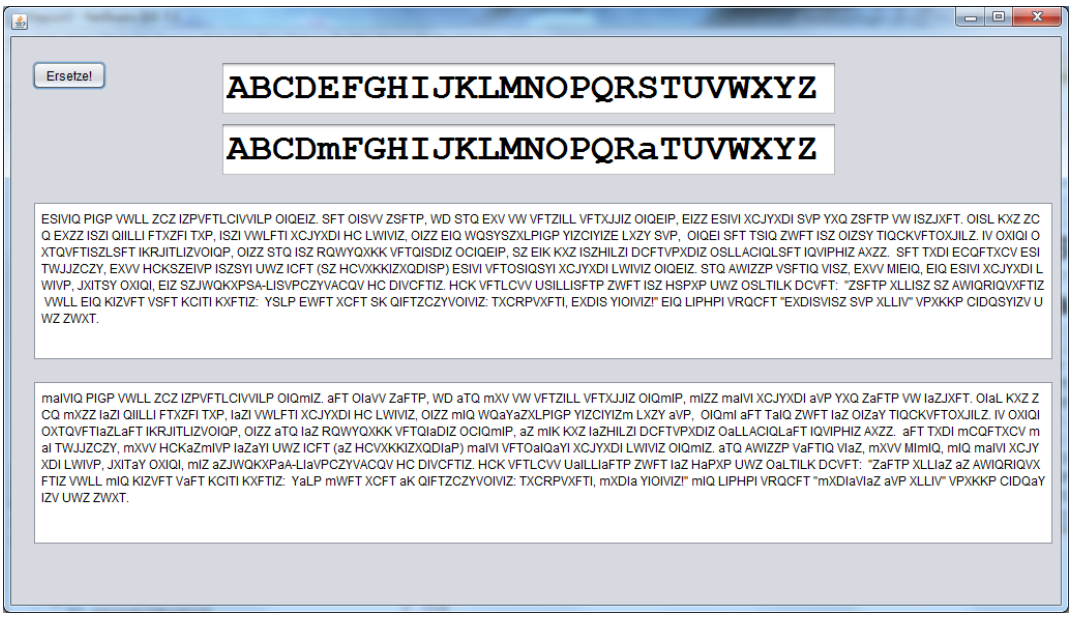
9. Verschlüsselungsverfahren: Alle Buchstaben eines Strings werden durch den zum Beispiel 3. Nachfolger im Alphabet ersetzt. Beispiel: ‚Uhrzeit' wird damit zu ‚Xkuchlw'. Beachte Groß- und Kleinschreibung. Benutze zur Lösung die ASCII-Codezahlen der Buchstaben! Schreibe ein Programm, welches den Originaltext in der Textarea *taEingabe* einliest und entsprechend kodiert in der Textarea *taAusgabe* wieder ausgibt!
Bemerkung: In einem längeren deutschen Text ist der Buchstabe ‚e' am häufigsten. Dies lässt sich benutzen, um einen derart verschlüsselten Text zu dekodieren.

10. Einfache Simulation einer Laufschrift.
Gegeben sei ein Label mit der Aufschrift „Informatik ist toll!"
Klickt man auf den Start-Button, so wird der erste Buchstabe dieses Textes gelöscht und dann hinten an den Text angehängt. Danach wird der neue Text auf dem Label ausgegeben. Wenn man gleichmäßig immer wieder auf den Start-Button klickt, erscheint so eine Laufschrift.
Hinweis: schöner wäre es natürlich, wenn die Laufschrift automatisch funktionieren würde. Dies ist in Java aber nur sehr umständlich zu realisieren. Probleme dabei sind die Wartezeiten und auch dass der sog. *sleep*-Befehl innerhalb von Schleifen nicht so, wie gewünscht, funktioniert (die Inschrift des Labels wird nicht dauernd aktualisiert.

11. Ein Text wird eingegeben. Der Rechner untersucht, ob dieser Text das **Wort** „ist“ bzw. „Ist“ enthält. Es gibt unterschiedliche Schwierigkeitsgrade:
„Das Essen ist gut.“ „Ist dies richtig?“ „Die Zahl ist entweder gerade oder sie ist ungerade.“ „In Istanbul war es heiß.“ „Bist du wach?“
Schreibe dein Programm zunächst möglichst einfach und verbessere es anschließend schrittweise.

12. Ein Satz wird eingegeben. Der Rechner meldet die Anzahl der Wörter in diesem Satz. Es gibt unterschiedliche Schwierigkeitsgrade:
a) der Satz enthält keine Kommata, Semikolons usw.
b) der Satz enthält auch beliebige Satzzeichen.

13. Ein verschlüsselter Text soll dekodiert werden. Es wird dabei davon ausgegangen, dass jeder Buchstabe durch einen bestimmten anderen Buchstaben kodiert worden ist. Betrachte dazu folgenden Screenshot:



Der verschlüsselte Text (aus Vereinfachungsgründen besteht er nur aus Großbuchstaben und Satzzeichen) steht in der oberen *Textarea*. In dem zweiten *Textfeld* kann der Benutzer einstellen, durch welchen (möglichst Klein-) Buchstaben der direkt oberhalb im ersten *Textfeld* stehende Großbuchstabe ersetzt werden soll. Bei Klick auf den Button *Ersetze!* wird diese Ersetzung durchgeführt. Indem man nach und nach einzelne Großbuchstaben durch die richtigen Kleinbuchstaben ersetzt, kann man schließlich den gesamten verschlüsselten Text dekodieren (in Kleinschrift).

Beide *Textareas* sollten die Eigenschaft *lineWrap* besitzen.
Außerdem sollten beide Textfelder den Schrifttyp *Courier New* (fett und Schriftgröße 26) besitzen, damit jeder Buchstabe gleich breit ist und so die Zuordnung der Buchstaben zueinander deutlicher wird.

Der verschlüsselte Text lautet:

```
ESIVIQ PIGP VWLL ZCZ IZPVFTLCIVVILP OIQEIZ. SFT OISVV ZSFTP, WD STQ EXV VW
VFTZILL VFTXJJIZ OIQEIP, EIZZ ESIVI XCJYXDI SVP YXQ ZSFTP VW ISZJXFT. OISL
KXZ ZCQ EXZZ ISZI QIILLI FTXZFI TXP, ISZI VWLFTI XCJYXDI HC LWIVIZ, OIZZ
EIQ WQSYSZXLPIGP YIZCIYIZE LXZY SVP,  OIQEI SFT TSIQ ZWFT ISZ OIZSY
TIQCKVFTOXJILZ.
IV OXIQI OXTQVFTISZLSFT IKRJITLIZVOIQP, OIZZ STQ ISZ RQWYQXKK VFTQISDIZ
OCIQEIP, SZ EIK KXZ ISZHILZI DCFTVPXDIZ OSLLACIQLSFT IQVIPHIZ AXZZ.
SFT TXDI ECQFTXCV ESI TWJJZCZY, EXVV HCKSZEIVP ISZSYI UWZ ICFT (SZ
HCVXKKIZXQDISP) ESIVI VFTOSIQSYI XCJYXDI LWIVIZ OIQEIZ. STQ AWIZZP VSFTIQ
VISZ, EXVV MIEIQ, EIQ ESIVI XCJYXDI LWIVP, JXITSY OXIQI, EIZ SZJWQKXPSA-
LISVPCZYVACQV HC DIVCFTIZ.
HCK VFTLCVV USILLISFTP ZWFT ISZ HSPXP UWZ OSLTILK DCVFT:
"ZSFTP XLLISZ SZ AWIQRIQVXFTIZ VWLL EIQ KIZVFT VSFT KCITI KXFTIZ:
 YSLP EWFT XCFT SK QIFTZCZYVOIVIZ: TXCRPVXFTI, EXDIS YIOIVIZ!"
EIQ LIPHPI VRQCFT "EXDISVISZ SVP XLLIV" VPXKKP CIDQSYIZV UWZ ZWXT.
```

Lösungen

Aufgabe 1

```
private void btStartMouseClicked(java…..) {
  String s = tfEingabe.getText();
  tfAusgabe.setText("" + s.length());
}
```

Aufgabe 2

```
private void btStartMouseClicked(java…..) {
  String s = tfEingabe.getText();
  s = s.toLowerCase();
  if (s.equals("goethe-gymnasium"))
    tfAusgabe.setText("akzeptiert");
  else   tfAusgabe.setText("nicht akzeptiert");
}
```

Aufgabe 3

```
private void btStartMouseClicked(java…..) {
  String s = tfEingabe.getText();
  int anzahl = 0;
  for (int i =0; i < s.length(); i++)
    if (s.charAt(i) == 'e') anzahl++;
  tfAusgabe.setText("" + anzahl);
}
```

Aufgabe 4

```
String  loescheZeichen(String s, int n) {
  return s.substring(0, n) + s.substring(n+1);
}
```

Aufgabe 5

```
private void btStartMouseClicked(java…..) {
  String s = tfEingabe.getText();
  int stelle;
  do {
    stelle = s.indexOf(" "); // Leerzeichen = (char) 32
    if (stelle >= 0) s=loescheZeichen(s, stelle);
  }
  while (stelle >= 0);
  tfAusgabe.setText(s);
}
```

Aufgabe 6

```
String  fuegeZeichenEin(String s, int n, char ch) {
  return s.substring(0, n) + ch + s.substring(n);
}

private void btStartMouseClicked(java…..) {
  String s = tfEingabe.getText();
  s = fuegeZeichenEin(s, 3, 'ü');
  tfAusgabe.setText(s);
}
```

Aufgabe 7

```
String ersetzeZeichenDurch(String s, int n, char ch) {
  return s.substring(0, n) + ch + s.substring(n+1);
}

private void btStartMouseClicked(java……) {
  String s = tfEingabe.getText();
  s = ersetzeZeichenDurch(s, 3, 'ü');
  tfAusgabe.setText(s);
}
```

Aufgabe 9

```
private void btStartMouseClicked(java…..) {
  String s = taEingabe.getText();
  char ch;
  int chNummer;
  int neueNummer;
  for (int i = 0; i < s.length(); i++) {
    ch = s.charAt(i);
    chNummer = (int) (ch);
    neueNummer = chNummer;
    if ((65 <= chNummer) && (91 >= chNummer ))
       neueNummer =  (chNummer - 65 +3)%26 + 65;
    else if ((97 <= chNummer) && (123 >= chNummer ))
           neueNummer =  (chNummer - 97 +3)%26 + 97;
    s = ersetzeZeichenDurch(s, i, (char) neueNummer);
  }
  taAusgabe.setText(s);
}
```

Aufgabe 10

```
private void btStartMouseClicked(java…..) {
  String s = labAusgabe.getText();
  char ch = s.charAt(0);
  s = loescheZeichen(s, 0) + ch;
  labAusgabe.setText(s);
}
```

Aufgabe 13

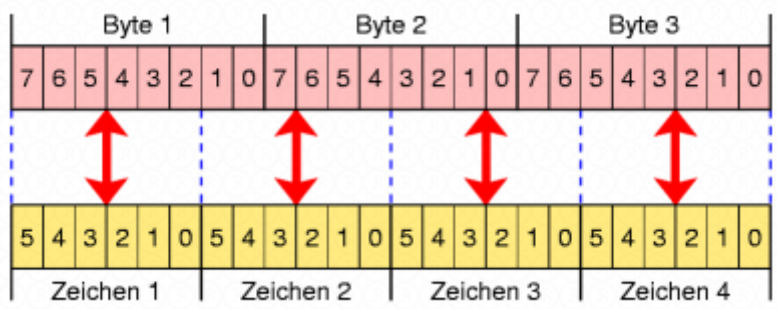
```
String ersetzeZeichenDurch(String s, int n, char ch) {
  return s.substring(0, n) + ch + s.substring(n+1);
}

private void btStartMouseClicked(java…..) {
  String alphabet = tfOriginal.getText();
  String codeAlphabet = tfDekodiert.getText();
  String originalText = taOriginal.getText();
```

```
  String dekodierterText = originalText;
  char ch;
  int stelle;
  for (int n = 0; n < originalText.length(); n++) {
    ch = originalText.charAt(n);
    stelle = alphabet.indexOf(ch);
    if (stelle >= 0)
      dekodierterText =
         ersetzeZeichenDurch(dekodierterText, n,
                        codeAlphabet.charAt(stelle));
  }
  taAusgabe.setText(dekodierterText);
}
```

Base64-Code

Base64 ist ein Verfahren zur Kodierung von beliebigen 8-Bit-Binärdateien (also Daten, die aus Bytes bestehen) in eine Zeichenfolge, die nur aus „druckbaren“ Zeichen besteht. Diese druckbaren Zeichen sind die Groß- und Kleinbuchstaben des Alphabets sowie die zehn Ziffern und zusätzlich noch die beiden Zeichen + und /.

Das *Base64*-Verfahren findet im Internet Anwendung und wird dort zum Beispiel zum Versenden von E-Mail-Anhängen (Bild- oder Musikdateien) verwendet. Der Grund dafür liegt darin, dass in E-Mail-Programmen nicht beliebige Bytes verschickt werden können, weil einige bestimmte Bytes eine besondere Bedeutung für den Datentransport haben. Allerdings werden die normalen Buchstaben und Ziffern grundsätzlich problemlos übertragen. Durch die Kodierung steigt der Platzbedarf des Datenstroms um (ziemlich genau) 33%.



Zur Kodierung werden jeweils drei Byte des Bytestroms (= 24 Bit) in vier 6-Bit-Blöcke aufgeteilt. Jeder dieser 6-Bit-Blöcke bildet eine Zahl von 0 bis 63. Diese Zahlen werden anhand der nachfolgenden Umsetzungstabelle in „druckbare *ASCII*-Zeichen“ umgewandelt und ausgegeben. Der Name des Algorithmus erklärt sich durch ebendiesen Umstand – jedem Zeichen des kodierten Datenstroms lässt sich eine Zahl von 0 bis 63 zuordnen (siehe Tabelle).

Falls die Gesamtanzahl der Eingabebytes nicht durch drei teilbar ist, wird die zu kodierende Eingabe am Ende mit aus Nullbits bestehenden Füllbytes aufgefüllt, sodass sich eine durch drei teilbare Anzahl an Bytes ergibt.

Hinweis: Auch bei bloßen Texteingaben ist die Anzahl der Eingabezeichen (mit Leer-, Satz- sowie Zeilensprungzeichen) nicht unbedingt identisch mit der Anzahl der Eingabebytes, weil z.B. deutsche Umlaute im utf-8-Zeichensatz durch zwei Bytes dargestellt werden.

Um dem Dekodierer mitzuteilen, wie viele Füllbytes angefügt wurden, werden diejenigen 6-Bit-Blöcke, die vollständig aus Füllbytes entstanden sind, mit dem

Zeichen = kodiert (also nicht durch den Buchstaben A, welcher normalerweise dem Sextett 000000 entspricht – siehe nachfolgende Codetabelle!). Somit können am Ende einer Base64-kodierten Datei null, ein oder zwei =-Zeichen auftreten. Anders gesagt, es werden so viele =-Zeichen angehängt, wie Füllbytes angefügt worden sind.

Bei einer zu kodierenden Eingabe mit Byte beträgt der Platzbedarf für den *Base64*-kodierten Inhalt $z = 4 \cdot \left\lceil \frac{n}{3} \right\rceil$ Zeichen. Die Klammern um den Bruch stehen für die aufrundende Ganzzahldivision.

Einfaches Beispiel:
Der Originaltext: „Info ist toll“ soll *Base64*-kodiert werden.

Die zugehörigen 13 Codezahlen der Zeichen sind dezimal:
73 110 102 111 32 105 115 116 32 116 111 108 108

Als achtstellige Binärzahlen hintereinander gereiht ergeben sich 13 Binärzahlen. Zusätzlich hängt man noch zwei Nullbytes an (weil 13 nicht durch 3 teilbar ist), so dass sich 15 Bytes ergeben:
0100 1001 | 0110 1110 | 0110 0110 | 0110 1111 | 0010 0000 | 0110 1001 |

0111 0011 | 0111 0100 | 0010 0000 | 0111 0100 | 0110 1111 | 0110 1100 |

0110 1100 | 0000 0000 | 0000 0000

Interpretiert man diese Folge von Nullen und Einsen als sechsstellige Binärzahlen, so erhält man folgende 20 (sechsstellige) Binärzahlen:
010010 | 010110 | 111001 | 100110 | 011011 | 110010 | 000001 | 101001 |

011100 | 110111 | 010000 | 100000 | 011101 | 000110 | 111101 | 101100 |

011011 | 000000 | 000000 | 000000
Dabei sind nur die beiden letzten Sextette vollständig aus Nullbytes entstanden. Diese beiden müssen also im Base64-Code durch zwei =-Zeichen kodiert werden.

Im Dezimalsystem entspricht das folgenden 20 Zahlen:
18 22 57 38 27 50 1 41 28 55 16 32 29 6 61 44 27 0 0 0

Laut der nachfolgenden Tabelle für den *Base64*-Zeichensatz entspricht das den folgenden 20 „druckbaren“ Zeichen:
SW5mbyBpc3QgdG9sbA==
(beachte, dass die letzten drei Nullen unterschiedlich kodiert werden!)

Base64-Zeichensatz

dez	binär		**dez.**	**binär**		**dez.**	**binär**		**dez.**	**binär**	
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

Weiteres Beispiel:
Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark

Der (mit Leerzeichen) 64 Zeichen lange Text ist *UTF-8*-kodiert 68 Byte lang, weil das ß und die Umlaute jeweils eine Länge von zwei Byte haben, und wird mit der Umwandlung in den *Base64*-Code zur folgenden 92 Zeichen langen *Base64*-Zeichenkette:

UG9seWZvbiB6d2l0c2NoZXJuZCBhw59lbiBNw6R4Y2hlbnMgVsO2Z2VsIFLDvGJlbiwgSm9naHVydCB1bmQgUXVhcms=

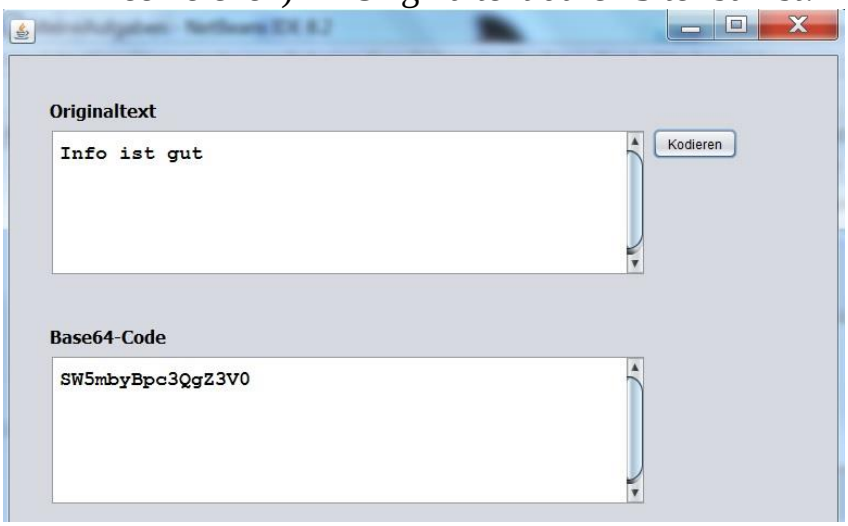
Aufgabe 1
Die Javamethode **`Integer.toBinaryString(int n)`** liefert (als String) die Binärdarstellung der Zahl n; allerdings nur mit minimal notwendiger Stellenzahl, also nicht unbedingt achtstellig.

Schreibe deshalb eine Methode **`String dualZahlwort(int n)`**, welche für den Parameter n mit $0 \leq n \leq 255$ eine genau achtstellige Dualzahl als String liefert! Oben erwähnte vordefinierte Javamethode sollte dabei genutzt werden.

Aufgabe 2
Schreibe eine Methode **`int binaer6ToDez(String dualzahl)`**, welche für eine 6-stellige Dualzahl (übergeben als Stringparameter) die zugehörige Dezimalzahl (vom Typ *int*) liefert!

Aufgabe 3
Schreibe eine Methode **`char base64Zeichen(int n)`**, welche für den Parameter n mit $0 \leq n \leq 63$ das zugehörige *base64*-Zeichen liefert!

Aufgabe 4
Schreibe ein Programm, welches zwei Textareas namens *taOriginal* und *taBase64* enthält. Der Benutzer kann einen beliebigen Text in *taOriginal* hineinschreiben und auf Button-Klick wird dieser entsprechend kodiert in der Textarea *taBase64* ausgegeben.
Tip: Setze zunächst voraus, dass die Anzahl der Eingabe-Bytes (einschließlich Leerzeichen) im Originaltext durch 3 teilbar ist!



Aufgabe 5
Wie Aufgabe 4, nur soll diesmal nicht kodiert sondern dekodiert werden. Benutze dafür einen zweiten Button!

Lösung Aufgabe 1

```
private String dualZahlwort(int n)  {
   String zahlwort = Integer.toBinaryString(n);
   int laenge = zahlwort.length();
   int differenz = 8 - laenge;
   for (int i=1; i<=differenz; i++) zahlwort = '0' + zahlwort;
   return zahlwort;
}
```

Lösung Aufgabe 2

```
private int binaer6ToDez(String dualzahl)  {
   int dezsum=0;
   if (dualzahl.charAt(0) == '1') dezsum = dezsum + 32;
   if (dualzahl.charAt(1) == '1') dezsum = dezsum + 16;
   if (dualzahl.charAt(2) == '1') dezsum = dezsum + 8;
   if (dualzahl.charAt(3) == '1') dezsum = dezsum + 4;
   if (dualzahl.charAt(4) == '1') dezsum = dezsum + 2;
   if (dualzahl.charAt(5) == '1') dezsum = dezsum + 1;
   return dezsum;
}
```

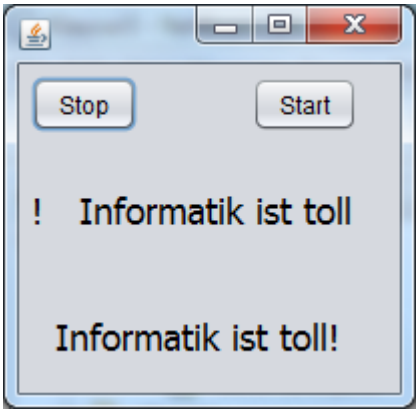
Lösung Aufgabe 3

```
private char base64Zeichen(int n){
   char ch;
   if (n <= 25) ch = (char) (n+65);
   else if (n <= 51) ch = (char) (n+71);
        else if (n <= 61) ch = (char) (n-4);
             else if (n == 62) ch = '+';
                  else ch = '/';
   return ch;
}
```

Lösung Aufgabe 4 *// mit einer durch 3 teilbaren Anzahl von Eingabe-Bytes*

```
private void jButton1MouseClicked(java….)  {
   String originalText = taOriginal.getText();
   int laenge = originalText.length(); // Vorsicht: Anzahl Bytes?
   String binaertext = "";
   for (int i = 0; i < laenge; i++)
      binaertext = binaertext + dualZahlwort((int) originalText.charAt(i));
   String base64Text = "";
   String sechsZiffern;
   for (int i=0; i < binaertext.length(); i=i+6)   {
        sechsZiffern = binaertext.substring(i, i+6);

        base64Text = base64Text + base64Zeichen(binaer6ToDez(sechsZiffern));
   }
   taBase64.setText(base64Text);
}
```

Timer



Das folgende Programm zeigt die Nutzung zweier *Timer*. Die Einzelheiten der Programmierung brauchen zum jetzigen Zeitpunkt noch nicht verstanden zu werden. Es geht allerdings aus dem Programmtext hervor, wie man *Timer* einsetzt. Das Programm zeigt zwei unterschiedlich schnelle Laufschriften. *Timer1* führt alle 500 ms die entsprechende Methode aus, *Timer2* nur jede Sekunde. Außerdem kann man die *Timer* ein- und ausschalten.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;

public class Laufschrift extends javax.swing.JFrame {
  Timer timer1;
  Timer timer2;

  public Laufschrift() {
    initComponents();
    initTimer1();
    initTimer2();
  }

  public void initTimer1()  {
    timer1 = new Timer(500, new ActionListener() {
           @Override
           public void actionPerformed(ActionEvent ae) {
             onTimer1();
           }
         } );
    timer1.start();
  }
```

```
public void onTimer1()  {
  String s = labAusgabe.getText();
  char ch = s.charAt(0);
  s = s.substring(1, s.length()) + ch;
  labAusgabe.setText(s);
}

public void initTimer2()  {
  timer2 = new Timer(1000, new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent ae) {
           onTimer2();
         }
        } );
  timer2.start();
}

public void onTimer2()  {
  String s = labAusgabe2.getText();
  char ch = s.charAt(0);
  s = s.substring(1, s.length()) + ch;
  labAusgabe2.setText(s);
}


private void btStopMouseClicked(java……) {
  timer1.stop();
  timer2.stop();
}

private void btStartMouseClicked(java……..) {
  timer1.start();
  timer2.start();
}
```

Die Klasse Math

Die Klasse *Math* aus dem Paket *java.lang* (welches automatisch immer in jedes Programm mit eingebunden wird, also nicht extra importiert werden muss) enthält mathematische Methoden und Konstanten. Alle Methoden und Konstanten dieser Klasse haben das Attribut *static* und können daher ohne konkretes Objekt verwendet werden. Beispiel: **double x = Math.sqrt(2)** Nachfolgend werden die wichtigsten von ihnen aufgelistet und eine kurze Beschreibung ihrer Funktionsweise gegeben.

Konstanten

PI entspricht der Kreiszahl π. Beispiel: double x = Math.PI

E entspricht der eulerschen Zahl e. Beispiel: double e = Math.E

Winkelfunktionen

Java.lang.Math stellt die üblichen Winkelfunktionen und ihre Umkehrungen zur Verfügung. Winkelwerte werden dabei **im Bogenmaß** übergeben.

```
public static double sin(double x)
public static double cos(double x)
public static double tan(double x)
public static double asin(double x)
public static double acos(double x)
public static double atan(double x)
```

Minimum und Maximum

Die Methoden *min* und *max* erwarten zwei numerische Werte als Argument und geben das kleinere bzw. größere von beiden zurück.

```
public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)

public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```

Arithmetik
Die nachfolgend aufgelisteten Methoden dienen zur Berechnung der Exponentialfunktion zur Basis *e*, zur Berechnung des **natürlichen** Logarithmus und zur Berechnung der Exponentialfunktion zu einer beliebigen Basis. Mit *sqrt* kann die Quadratwurzel berechnet werden.

```
public static double exp(double a)
public static double log(double a)
public static double pow(double a, double b)
public static double sqrt(double a)
```

Runden und Abschneiden
Mit Hilfe der Methode *abs* wird der absolute Betrag eines numerischen Werts bestimmt, *ceil* liefert die kleinste ganze Zahl größer und *floor* die größte ganze Zahl kleiner oder gleich dem übergebenen Argument. Mit Hilfe von *round* kann ein Wert gerundet werden.

```
public static int abs(int a)
public static long abs(long a)
public static float abs(float a)
public static double abs(double a)

public static double ceil(double a)
public static double floor(double a)
public static long round(float a)
```

Zufallszahlen

Eine weitere Funktion der Klasse *Math* stellt reelle Zufallszahlen vom Typ *double* zur Verfügung. Die Anweisung *double zahl = Math.random();* liefert eine zufällige reelle Zahl aus dem Intervall [0; 1[, also einschließlich 0 und ausschließlich 1.

Möchte man eine reelle Zufallszahl aus dem Intervall [0; 15[erzeugen, so schreibt man: *zahl = Math.random() * 15;*
Eine zufällige reelle Zahl aus dem Intervall [14; 18[erhält man durch die Anweisung *zahl = Math.random() * 4 + 14;*

Möchte man eine zufällige ganze Zahl haben, so ist eine Typumwandlung notwendig. Diese geschieht durch Voranstellen des gewünschten Typs in Klammern: *int ganzZahl = (int) (Math.random() * 4 + 14);*
oder durch *long ganzZahl = Math.round(Math.random() * 4 + 14);*

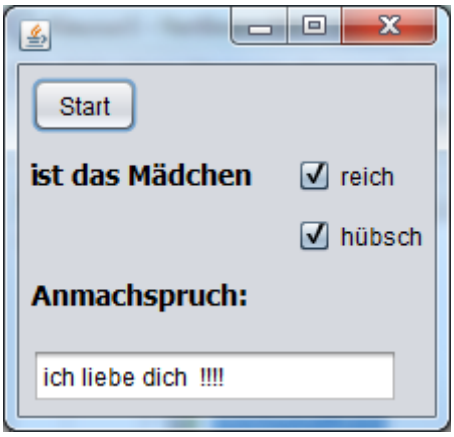
Beachte die Klammersetzung bei der Generierung der Zufallszahl! Diese ist wegen der Prioritätenregelung *Funktion vor Punktrechnung vor Strichrechnung* notwendig. Die Anweisung *ganzZahl = (int) Math.random() * 4 + 14;* würde immer das Ergebnis 14 liefern.

Aufgaben

1. Erstelle ein Programm, mit dessen Hilfe man Winkelumrechnungen vornehmen kann. Eingegeben wird ein Winkel α im Gradmaß, ausgegeben der zugehörige Winkel x im Bogenmaß. Analog auch anders herum.
Hinweis: es gilt $\frac{\alpha}{360°} = \frac{x}{2 \cdot \pi}$

2. Programmiere einen Rechentrainer für das kleine Einmaleins! Der Rechner stellt eine zufällige Aufgabe, der Benutzer gibt das Ergebnis ein, welches vom Rechner überprüft wird. Der Rechner gibt die Meldung „falsch“ oder „richtig“ aus.
Dies kann der Benutzer beliebig oft wiederholen. Die Anzahl der Fehler bzw. richtigen Eingaben wird mitgezählt. Nach jeder Benutzereingabe wird die Anzahl der richtigen und falschen Lösungen ausgegeben.

Checkboxen



Als Beispiel für die Benutzung von sog. Checkboxen diene folgendes Programm:

```
private void btStartMouseClicked(java…..) {
   if (cb1.isSelected()) tfAusgabe.setText("ich liebe
                                          dich  !!!!");
   else if (cb2.isSelected()) tfAusgabe.setText
                                        ("Hey Baby !");
        else tfAusgabe.setText("" + (char) 9786);
}
```

Aufgabe

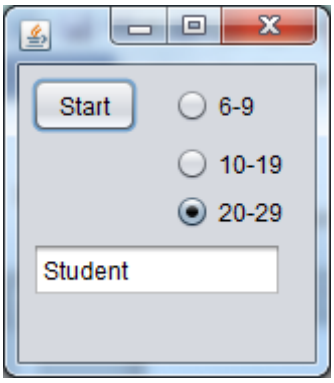
1. Erstelle ein Bewerbungsformular für eine Firma. Die Firma möchte wissen, ob der Bewerber 10 bestimmte Fähigkeiten hat. Diese Fähigkeiten werden mit Checkboxen abgefragt (ich habe einen Führerschein; ich verstehe englisch, ich kann schwimmen, usw.). Die Anzahl der Haken an den Checkboxen entscheidet über die Ausgabe des Ergebnisses („sehr geeignet“, „evtl. geeignet“, „ungeeignet“).

Radio-Buttons

Radiobuttons kann man entweder einzeln benutzen (wie eine Checkbox) oder sie einer **Radiobutton-Gruppe** zuordnen. Innerhalb einer Gruppe kann immer nur ein einziger Radiobutton aktiviert sein.
Natürlich kann man auch mehrere Gruppen von Radiobuttons einsetzen.

Möchte man eine Gruppe von Radiobuttons erzeugen, so zieht man in NetBeans zuerst eine sog. *Button-Group* auf das Formblatt und gibt dieser Gruppe einen sinnvollen Namen.
Danach zieht man mehrere Buttons auf das Formblatt. Im Eigenschaftenfenster eines Radiobuttons kann man diesen einer Gruppe zuordnen. Außerdem kann hier auch (logischerweise nur) einem dieser Gruppenbuttons die Eigenschaft *Selected* gegeben werden.

Den Wert eines Radiobuttons fragt man mit *isSelected()* ab. Die Funktion gibt einen boolschen Wert zurück (*true* oder *false*), man kann die Funktion also direkt in einer *if*-Abfrage verwenden.




```
private void btStartMouseClicked(java……..) {
   if (rb6_9.isSelected())
                    tfAusgabe.setText("Grundschüler");
   else if (rb10_19.isSelected())
                tfAusgabe.setText("Gymnasialschüler");
        else if (rb20_29.isSelected())
                        tfAusgabe.setText("Student");
}
```

Aufgabe

1. Programmiere einen Rechentrainer. Der Rechner stellt eine zufällige Aufgabe mit natürlichen Zahlen, der Benutzer gibt das Ergebnis ein, welches vom Rechner überprüft wird. Der Rechner gibt die Meldung „falsch“ oder „richtig“ aus. Die Aufgabenart (Addition, Subtraktion, Division, Multiplikation, Potenzrechnung, Zufallsrechenart) wird durch die Anwahl eines von 6 Radiobuttons bestimmt. Bei der Subtraktion ist darauf zu achten, dass das Ergebnis nicht negativ wird. Bei Divisionsaufgaben soll das Ergebnis ganzzahlig sein.

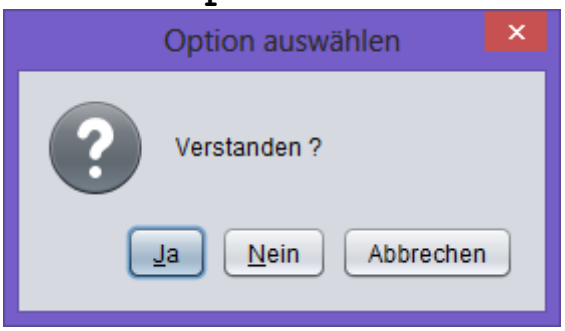
Dialog-Fenster

Mithilfe von Klassenmethoden der Klasse *JOptionPane* lassen sich unterschiedliche Dialogfenster einsetzen. **Voraussetzung: es wird das Package *javax.swing.*;* importiert.**
Eine einfache Textausgabe erhält man dann durch die folgende Anweisung:

```
JOptionPane.showMessageDialog(this, "Der Rechner explodiert gleich");
```

Eine simple Ja-Nein-Abfrage erhält man so:

```
int wahl;
wahl = JOptionPane.showConfirmDialog(this , <text>);
```

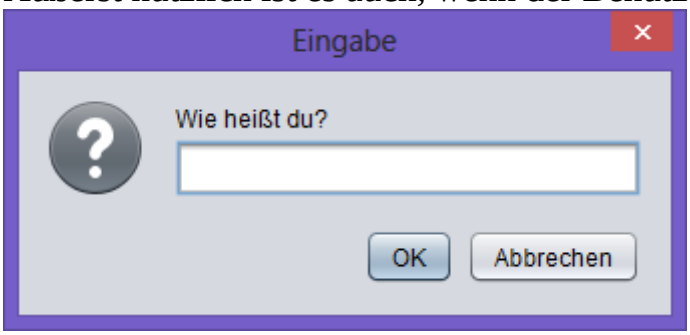


Diese Dialogbox liefert eine Zahl zurück und zwar:

Der Rückgabewert ist 0 für Ja, 1 für Nein, 2 für Abbrechen und -1, wenn das Dialogfenster zugemacht wurde (mit dem Kreuz oben rechts am Fensterrand).
<text> kann ein Klartext in Anführungszeichen sein oder eine String-Variable.
Das *this* muss genau so da stehen.

```
Beispiel:
int wahl = JOptionPane.showConfirmDialog(this ,
"Verstanden ?");
if (wahl == 0){
   JOptionPane.showMessageDialog( this, "Wenn Sie 'Ja'
                        sagen, warum sind Sie dann immer noch da?");
}
```

Äußerst nützlich ist es auch, wenn der Benutzer einen String eingeben kann:




```
String name;
name = JOptionPane.showInputDialog(this, "Wie heißt du?");
JOptionPane.showMessageDialog(this, "Hallo "+name+",
                              du darfst nicht spielen!");
int wahl = JOptionPane.showConfirmDialog(this ,
                                         "Verstanden?");
```

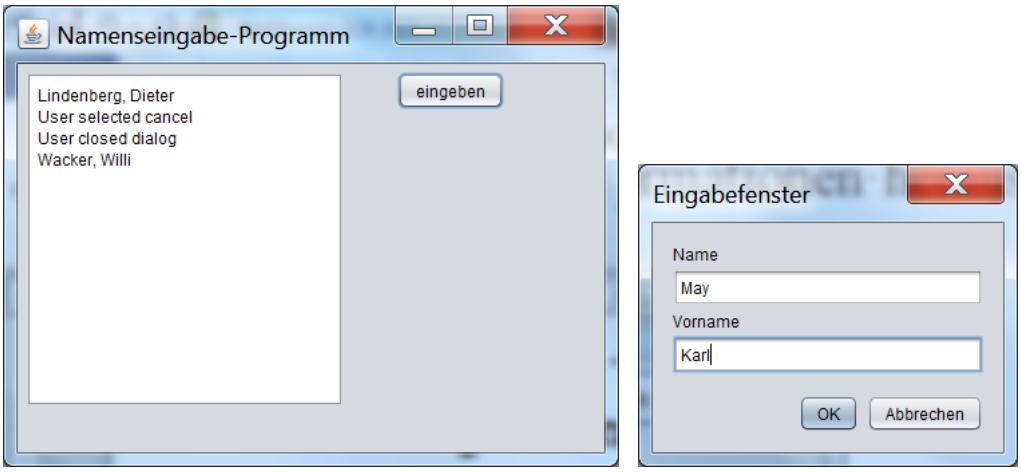
Aufgaben

1. Der Benutzer wird (natürlich über ein Dialog-Fenster) aufgefordert, eine natürliche Zahl (es wird vorausgesetzt, dass diese Zahl noch im Integer-Bereich liegt) einzugeben. Der Rechner meldet anschließend nach der Eingabe, ob die eingegebene Zahl gerade oder ungerade ist.

2. Der Benutzer wird gefragt, ob er das Fach Informatik interessant findet oder nicht. Nach seiner Eingabe reagiert der Rechner mit einer entsprechenden Meldung.

3. Dem Benutzer wird die Aufgabe „Wieviel ist 3·4 ?“ gestellt. Nach seiner Eingabe wird diese Eingabe ausgewertet und es wird dem Benutzer eine entsprechende Rückmeldung (richtig oder falsch) gegeben.

Im Folgenden wird ein selbstgestaltetes Eingabe-Fenster erstellt, aus dem man mehrere (im Beispiel zwei) Informationen herausholen kann.




```
import javax.swing.JOptionPane;
import javax.swing.JTextField;

public class EigenerDialog extends javax.swing.JFrame
{

    public EigenerDialog() {
        initComponents();
    }

private void btEingebenMouseClicked(java…..) {
  JTextField nameFeld = new JTextField();
  JTextField vornameFeld = new JTextField();
  // Erstellung Array vom Datentyp Object, Hinzufügen der Komponenten
  Object[] EingabeDialog = {"Name", nameFeld, "Vorname",
                                                    vornameFeld};

  JOptionPane pane = new JOptionPane(EingabeDialog,
                                JOptionPane.PLAIN_MESSAGE,
                                 JOptionPane.OK_CANCEL_OPTION);
  pane.createDialog(null, "Eingabefenster").setVisible(true);

  if(null == pane.getValue()) taAusgabe.append("User closed
                                                    dialog\n");
  else {
    switch((int) pane.getValue()) {
      case JOptionPane.OK_OPTION:
           taAusgabe.append(nameFeld.getText()+ ", " +
                                 vornameFeld.getText()+ "\n" );
          break;
      case JOptionPane.CANCEL_OPTION:
           taAusgabe.append("User selected cancel\n");
           break;
```

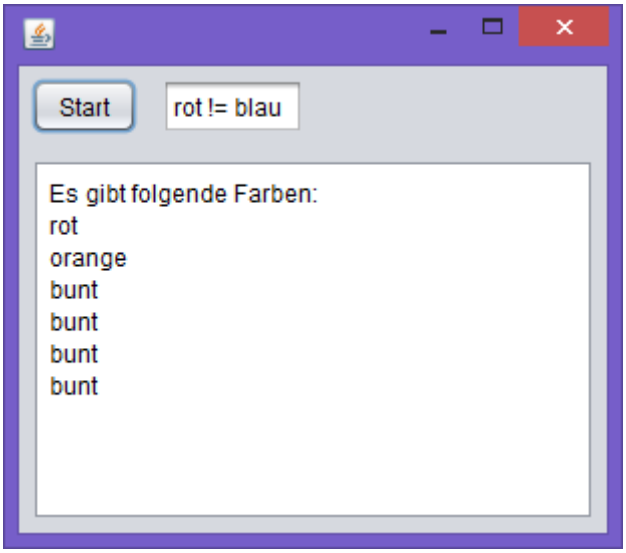
```
        default:
      taAusgabe.append("User selected "+pane.getValue() + "\n");
      } // end of switch
    } // end of else
  } // end of btEingebenMouseClicked

} // end of class
```

Aufgabe

1. Ändere obiges Programm so, dass mehr als nur zwei sinnvolle Eingaben im Eingabefenster eingetragen werden müssen. All diese eingetragenen Informationen sollen im Hauptfenster in der Textarea ausgegeben werden.

Aufzählungstypen

Unter einfachen Aufzählungen versteht man die Zusammenfassung von mehreren konstanten Werten zu einer Einheit. Die Variablen solcher Aufzählungen sind dann nur in der Lage, diese vordefinierten konstanten Werte zu beinhalten.
Ein Aufzählungstyp wird in Java mit ***enum*** bezeichnet.




```
public enum Farben {ROT, ORANGE, GELB, GRUEN, BLAU,
VIOLETT}


public void farbAusgabe(Farben f) {
  String s;
  switch (f)  {
  // in switch-Anweisungen wird der Typ nicht mit angegeben sondern nur der
  // entsprechende Wert, also nicht Farben.ROT sondern nur ROT
      case ROT: s = "rot"; break;
      case ORANGE: s = "orange"; break;
      default: s= "bunt";
  }
  taAusgabe.append(s + "\n");
}
```

```
private void btStartMouseClicked(java……) {
  Farben f1 = Farben.ROT;
  if (f1 == Farben.BLAU) tfAusgabe.setText("rot = blau");
  else tfAusgabe.setText("rot != blau");

  taAusgabe.append("Es gibt folgende Farben:\n");

  for (Farben f: Farben.values())  farbAusgabe(f);
}
```

Bemerkung:
die einzelnen Werte eines Aufzählungstyps sind nach der bei der Deklaration festgelegten Reihenfolge geordnet. Die Methode ***f1.compareTo(f2)*** vergleicht die Werte der beiden Aufzählungsvariablen *f1* und *f2* und gibt entweder -1, 0 oder 1 zurück, je nachdem ob *f1* eher oder später in der Reihenfolge vorkommt.

Aufgabe
Es sollen alle Karten eines Skatspiels in einer Textarea ausgegeben werden, also:
Karo Sieben
Karo Acht
….
Kreuz Ass

Benutze dazu die beiden Aufzählungstypen *Farbe = {Karo, Herz, Pik, Kreuz}* und *Bild = {Sieben, Acht, Neun, Zehn, Bube, Dame, König, Ass}*

Modifier

Schlüsselwörter wie z.B. *public*, *private* oder *protected* werden als *Modifier* bezeichnet. Mit Hilfe dieser Attribute können die Eigenschaften von Klassen, Methoden und Variablen festgelegt werden. Sie haben insbesondere Einfluss auf die *Lebensdauer*, *Sichtbarkeit* und *Veränderbarkeit* dieser Programmelemente.

- Elemente des Typs *public* sind in der Klasse selbst (also in ihren Methoden), in Methoden abgeleiteter Klassen und für den Aufrufer von Instanzen der Klasse sichtbar.
- Elemente des Typs *protected* sind in der Klasse selbst und in Methoden abgeleiteter Klassen sichtbar. Zusätzlich können Klassen desselben Pakets sie aufrufen.
- Elemente des Typs *private* sind lediglich in der Klasse selbst sichtbar. Für abgeleitete Klassen und für Aufrufer von Instanzen bleiben private-Variablen verdeckt.
- Elemente, die ohne einen der drei genannten *Modifier* deklariert wurden, werden als *package scoped* oder Elemente mit *Standard-Sichtbarkeit* bezeichnet. Sie sind nur innerhalb des Pakets sichtbar, zu dem diese Klasse gehört. In anderen Paketen sind sie dagegen unsichtbar. Elemente mit Standard-Sichtbarkeit verhalten sich innerhalb des Pakets wie *public*- und außerhalb wie *private*-Elemente.

Bemerkungen:
Die Einschränkung *private* bedeutet überraschenderweise nicht, dass die Methoden einer Klasse nur auf die privaten Variablen des eigenen Objekts zugreifen dürfen. Vielmehr ist ebenfalls möglich, (quasi von außen) auf die private-Variablen eines *anderen* Objekts derselben Klasse zuzugreifen.

Das Attribut *public* ist zusätzlich auch bei der Klassendefinition selbst von Bedeutung, denn nur Klassen, die als *public* deklariert wurden, sind außerhalb des Pakets sichtbar, in dem sie definiert wurden**. In jeder Quelldatei darf nur eine Klasse mit dem Attribut *public* angelegt werden.**

In einer Quelldatei kann es immer nur eine Hauptklasse geben, die so wie die Datei heißt. Weitere Klassen innerhalb der Quelldatei dürfen nicht *public* sein.

static

Variablen und Methoden mit dem Attribut *static* sind nicht an die Existenz eines konkreten Objekts gebunden, sondern existieren vom Laden der Klasse bis zum Beenden des Programms. Das *static*-Attribut beeinflusst bei Variablen ihre Lebensdauer und erlaubt bei Methoden den Aufruf, ohne dass der Aufrufer ein Objekt der Klasse besitzt, in der die Methode definiert wurde.

Wird das Attribut *static* nicht verwendet, so sind Variablen innerhalb einer Klasse immer an eine konkrete Instanz gebunden. Ihre Lebensdauer beginnt mit dem Anlegen des Objekts und dem Aufruf eines Konstruktors und endet mit der Freigabe des Objekts durch den Garbage Collector.

Beispiel: die Klasse *Math* aus dem Paket java.lang enthält ausschließlich statische Methoden. Diese werden durch Voranstellen des Klassennamens aufgerufen: double x = Math.sqrt(2)

final

Membervariablen mit dem Attribut *final* dürfen nicht verändert werden, sind also als *Konstanten* anzusehen. Methoden des Typs *final* dürfen nicht überlagert werden; ebensowenig dürfen Klassen des Typs *final* zur Ableitung neuer Klassen verwendet werden. Wird das Attribut *final* dagegen nicht verwendet, sind Membervariablen veränderbar, können Methoden überlagert und Klassen abgeleitet werden.

Vererbung

```
class Basis {
  public Basis() {
    System.out.println("es wurde gerade der
            Konstruktor der Basisklasse aufgerufen");
    // Textareas besitzt diese Klasse nicht
  }
}


class Erbe extends Basis {
  public Erbe() {
    System.out.println("es wurde gerade der
            Konstruktor der Unterklasse aufgerufen");
    // Textareas besitzt diese Klasse nicht
  }
}


public class Versuch2 extends javax.swing.JFrame {
  public Versuch2() {
    initComponents();
    Erbe x = new Erbe();
  }
…..
}
```

Wird dieses Programm gestartet, so erhält man folgende Ausgabe:

es wurde gerade der Konstruktor der Basisklasse aufgerufen
es wurde gerade der Konstruktor der Unterklasse aufgerufen

An dieser Ausgabe ist erkennbar, dass bei der Erzeugung eines Objektes der Klasse *Erbe* nicht nur der Standardkonstruktor der Klasse *Erbe* , sondern auch der Standardkonstruktor der Basisklasse aufgerufen wird – und zwar zuerst der Basisklassenkonstruktor.

Die oberste Klasse in Java ist die Klasse *Object*. Alle anderen Klassen sind Unterklassen von *Object*, ohne dass dies extra angegeben werden muss. Deshalb ist auch immer folgende Zuweisung möglich:

```
Object x;
Erbe Willi = new Erbe();
x = Willi;
```

Polymorphismus

```
class Person  {
  protected String name;
  public Person()  {
     name = "Personenname";
  }

  public void ausgabe()  {
     System.out.println("Name: " + name);
     System.out.println();
  }

  public void ausfuehrlicheAusgabe()  {
     System.out.println("mein Personenname: " + name);
     System.out.println();
  }
}


class Kunde extends Person  {
  private int kundenID;
  public Kunde()  {
     super();  // überflüssig, weil dieser Aufruf automatisch erfolgt
     kundenID = 0;
  }

  public void kAusgabe()  {
     System.out.println("Name: " + name);
     System.out.println("Kunden-ID: " + kundenID);
     System.out.println();
  }

  public void ausfuehrlicheAusgabe()  {
     System.out.println("mein Kundenname: " + name);
     System.out.println();
  }
}
```

```
public class Versuch3 extends javax.swing.JFrame {
   public Versuch3() {
      initComponents();
   }

.....

  private void btStartMouseClicked(java.....) {
    Person einePerson;
    Kunde einKunde = new Kunde();

    einKunde.ausgabe();
    einKunde.kAusgabe();

    einePerson = einKunde;
    einePerson.ausgabe();
    einePerson.ausfuehrlicheAusgabe();
    //einePerson.kAusgabe();  nicht möglich
  }
} // Ende der Klasse Versuch3
```

Obiges Programm bewirkt folgende Ausgaben:

Name: Personenname

Name: Personenname
Kunden-ID: 0

Name: Personenname

mein Kundenname: Personenname

Innerhalb einer Vererbungshierarchie können durchaus Objektvariablen der Oberklasse (von welcher Klasse die Variable ist, wird durch die Deklaration festgelegt!) Objekte der Unterklasse beinhalten bzw. zugewiesen werden. Das ist sinnvoll, denn die Objekte der Unterklasse besitzen alle öffentlichen Attribute und Methoden, welche ein Objekt der Oberklasse haben muss. Hinweis: Auf private Attribute darf ein **Objekt** sowieso nicht zugreifen, das darf

nur die eigene **Klasse**.

Umgekehrt darf man einer Objektvariablen der Unterklasse (wiederum festgelegt durch die Deklaration) keine Instanz der Oberklasse zuordnen. Begründung: Objekte der Unterklasse haben üblicherweise mehr *public* Methoden als die Oberklasse. Bei einer Objektvariablen, die per Deklaration der Unterklasse zugeordnet worden ist, sollte man auch alle *public* Methoden der Unterklasse aufrufen können.

In der Klasse *Person* gibt es die Methode *ausfuehrlicheAusgabe()*, welche in der Unterklasse *Kunde* überschrieben (neu implementiert) wird. Man nennt diese Methoden auch *virtuell* In anderen Programmiersprachen müssen *virtuelle* Methoden auch gesondert gekennzeichnet werden – in Java ist das nicht nötig.

Interessant ist folgendes Problem: Nachdem man der Objektvariablen *einePerson* der Oberklasse die Instanz *einKunde* der Unterklasse zugewiesen hat, wird bei der Anweisung *einePerson.ausfuehrlicheAusgabe()* die entsprechende Unterklassenmethode ausgeführt.

Diese Vorgehensweise nennt man ***Polymorphie*** oder auch ***dynamische*** oder ***späte Bindung*** von Methoden. Erst zur Laufzeit des Programms wird geprüft, on welcher Klasse (hier: *Person* oder *Kunde*) das Objekt zum Zeitpunkt des Aufrufes ist, und in Abhängigkeit von dieser Prüfung wird entschieden, welche der beiden gleichnamigen Methoden *ausfuehrlicheAusgabe()* ausgeführt wird. Diese Prüfung findet natürlich nur für virtuelle bzw. überschriebene Methoden statt.
Das Ergebnis dieser Prüfung ist übrigens nicht abhängig von der Deklaration der Typvariablen (bekanntlich kann eine Variable der Oberklasse auch ein Objekt einer Unterklasse beinhalten).

Dieselbe Prüfung findet nicht nur statt, wenn die Methode *ausfuehrlicheAusgabe()* direkt als Objektmethode aufgerufen wird, sondern auch dann, wenn sie indirekt als Teil innerhalb einer anderen Methode aufgerufen wird.

Seit der Version Java 5 gibt es die sog. Annotationen, die der Programmierer neben den Java-Befehlen und Kommentaren im Quelltext einfügen kann. Diese

Annotationen sollen dem Compiler bestimmte Absichten des Programmierers mitteilen. Wenn beispielsweise eine Methode überschrieben werden soll, dann kann das durch die Annotation @Override kenntlich gemacht werden. Der Compiler prüft daraufhin, ob die Überschreibung syntaktisch korrekt ist.

Im Quelltext steht dann diese Annotation direkt vor dem Methodennamen:

```
@Override
public void ausfuehrlicheAusgabe()    {
.....
}
```

Arrays

Oft hat man mehrere oder sogar sehr viele Variabeln vom selben Datentyp, die irgendwie zusammen gehören.
Beispiele: 900 Namen (vom Typ *String*), welche den Schülern einer Schule entsprechen. Oder 10 mündliche Noten (vom Typ *int*), die zu einem bestimmten Schüler im Fach XY gehören. Oder die Breiten- und Längengrade von allen Hauptstädten der Erde. Oder die drei Ortskoordinaten (vom Typ *double*) von mathematischen Punkten im Raum.

Solche zusammengehörigen Daten kann man in Feldern (englisch: Arrays) zusammenfassen.

Im Folgenden wird als Beispiel für 10 zusammengehörende Noten ein Feld erstellt:
Zunächst wird eine entsprechende Feldvariable namens Notenfeld **deklariert**:
```
int[] Notenfeld;
```
Dabei steht noch nicht fest, wie viele Elemente dieses Feld haben soll.
Später kann man dieses Feld **erzeugen**: `Notenfeld = new int[10];`

Damit steht fest, dass ein Notenfeld mit 10 Elementen (nummeriert von 0 bis 9) existiert. Allerdings sind die Werte dieser 10 Elemente noch nicht festgelegt.
Die Deklaration und das Erzeugen wird oft in einem einzigen Schritt erledigt:
```
int[] Notenfeld = new int[10];
```

Will man später die 10 Elemente mit Anfangswerten belegen, also das Feld **initialisieren**, so wäre das zum Beispiel folgendermaßen möglich:
```
for (int i = 0; i < 10; i++)  Notenfeld[i] = i*i;
```

Achtung:
Obiges Notenfeld besitzt zwar 10 Elemente, aber der Index *i* läuft von 0 bis 9.

Nur bei der Deklaration eines Arrays kann auch sofort eine Initialisierung erfolgen, indem direkt die Werte für das Array in geschweiften Klammern angegeben werden. Die Länge bzw. Tiefe des Arrays ergibt sich damit automatisch aus der Anzahl der Werte in den geschweiften Klammern:
```
int[] feld = {1, 4, 9, 16, 25, 36};
```

Allgemein (also für Felder, welche beliebige Datentypen enthalten) kann die Deklaration und Erzeugung eines Arrays so dargestellt werden:

```
Datentyp[]  Bezeichner = new Datentyp[Anzahl]
```
Beispiel: `STRING[]  namensliste = new STRING[900];`

Die for each – Schleife

Diese Schleife ist speziell für Arrays konzipiert worden. Mit ihrer Hilfe können **die Werte** eines Arrays sehr einfach durchlaufen werden, ohne dass bekannt sein muss, wie viele Elemente das Array hat.

Syntax der for each-Schleife:
`for (Datentyp wert: Arrayname)  {…}`
Dabei muss das Array natürlich Elemente von diesem Datentyp enthalten.
Wichtig: *wert* ist dabei **keine Laufvariable** (0..Arrayname.length – 1), sondern enthält alle Werte des Arrays.
Beispiel: `int[] Zahlenfeld = {1, 4, 9, 16, 25, 36};`

```
for (int wert: Zahlenfeld) System.out.println(wert);
```
Durch diese Anweisung werden obige 6 Quadratzahlen (und nicht die Zahlen 0 bis 5) ausgedruckt!

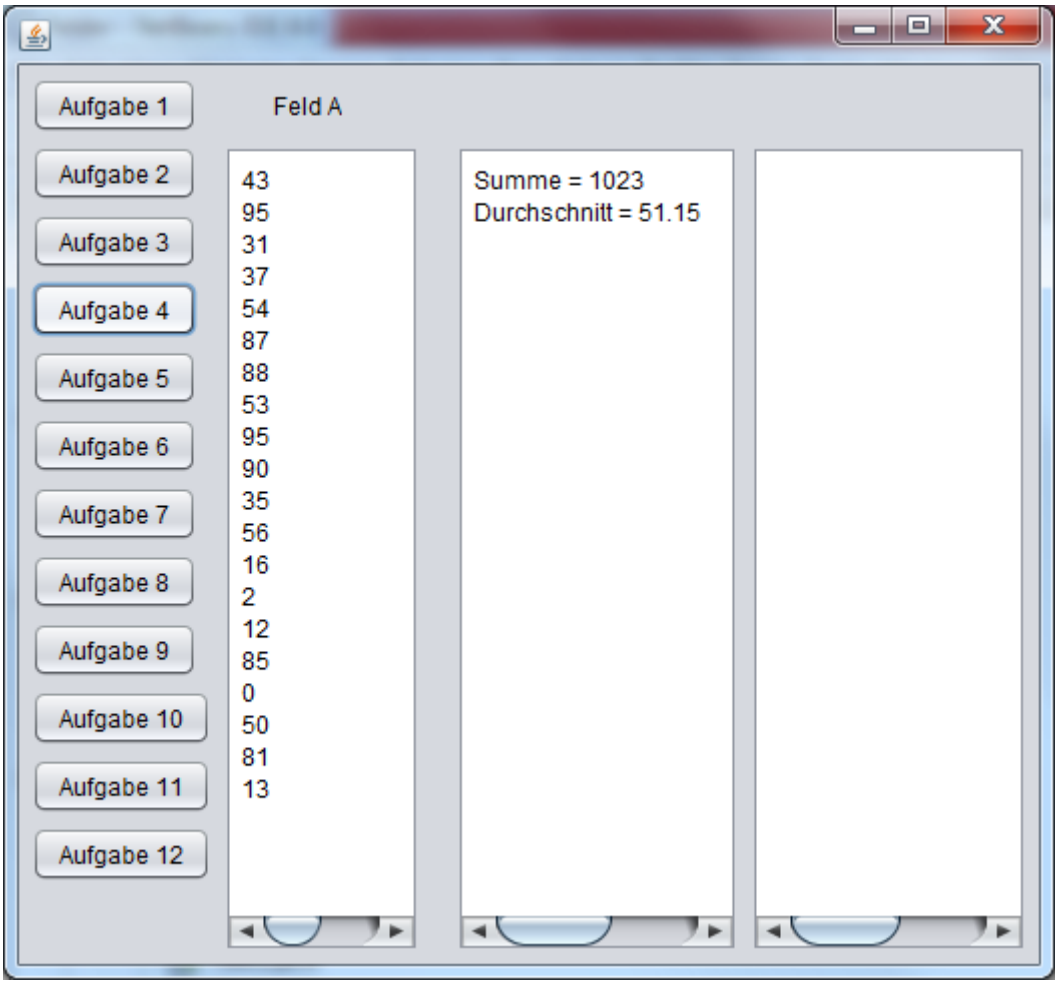
Die letzte Anweisung bewirkt dasselbe wie die folgende:
```
for (int i = 0; i < Zahlenfeld.length; i++)
                        System.out.println(Zahlenfeld[i]);
```

Beachte, dass die nachfolgende Anweisung nur die Werte der Laufvariablen *i* ausgibt:
```
for (int i = 0; i < Zahlenfeld.length; i++)
                                    System.out.println(i);
```

Beispiel für das Kopieren eines Arrays:
```
int[] A = {1, 5, 7};
int[] B = new int [3];
for (int i = 0; i < A.length; i++)  B[i] = A[i];
```

Aufgaben mit Arrays

Benutze für das folgende Programm ein Integer-Array mit dem Namen *A*, welches global deklariert wird, aber erst in den einzelnen Aufgaben-Methoden erzeugt wird.
Der folgende Screenshot zeigt eine mögliche Lösung der Aufgabe 4:



1. Das Feld A enthält zufällige, natürliche Zahlen. Alle Zahlen sollen (unsortiert) in einer Textarea ausgegeben werden.
   a) vorwärts b) rückwärts

2. Das Feld A soll die ersten 100 Quadratzahlen enthalten (also 1, 4, 9, 16, … , 10 000). Zur Kontrolle werden alle Zahlen des Arrays ausgegeben.

3. Das Feld A enthält zufällige positive Zahlen, die alle kleiner als 1000 sind. Die kleinste Zahl soll ausgegeben werden. Außerdem soll die Nummer der kleinsten Zahl im Feld mit ausgegeben werden. Beispiel: A[11] enthält die kleinste Zahl 3. Dann sollen sowohl 3 als auch 11 ausgegeben werden. Mache anschließend dasselbe mit der größten Zahl des Feldes.
Hinweis: In Java hat die größte Integer-Zahl den Namen *Integer.MAX_VALUE* und die kleinste (negative) Integer-Zahl hat den Namen *Integer.MIN_VALUE*.

4. Das Feld A enthält zufällige positive Zahlen. Die Summe aller Zahlen im Array wird ausgegeben. Gib auch den Durchschnittswert aller Zahlen aus!

5. Das Feld A soll zufällige Zahlen enthalten. Es wird zur Kontrolle ausgegeben. Alle geraden Zahlen des Zufallfeldes A sollen in der nebenstehenden Textarea ausgegeben werden.

6. Es wird ein zufälliges Zahlenfeld A erzeugt und in einer Textarea ausgegeben. Danach wird der Benutzer (mit einem Dialog-Fenster) aufgefordert, eine beliebige Zahl einzugeben. Der Rechner überprüft daraufhin, ob sich diese Zahl in dem Array A befindet oder nicht. Eine entsprechende Dialog-Meldung wird ausgegeben.

7. Gegeben sind zwei Arrays A und B, welche jeweils 30 Zufallszahlen (kleiner als 60) enthalten. Die beiden Felder A und B haben nicht identischen Inhalt. In zwei Textareas werden diese Zahlen ausgegeben. In einer dritten Textarea werden alle Zahlen ausgegeben, die sowohl im Feld A als auch im Feld B enthalten sind.

8. Erhöhe sämtliche Zahlen aus dem Feld A um 1 und gib anschließend das neue Feld A in der nebenstehenden Textarea aus!

9. Verschiebe alle Zahlen aus dem Feld A um eine Stelle nach rechts (im Feld A). Die letzte Zahl aus dem alten Feld A soll die erste Zahl im neuen Feld A werden. Gib anschließend das neue Feld A aus!

10. Es sollen alle Primzahlen, die kleiner als 1000 sind, in einer Textarea ausgegeben werden.
Bemerkung: Primzahlen sind alle natürlichen Zahlen, die größer sind als 1 und die nur durch sich selbst und durch 1 ohne Rest teilbar sind.
Lösungsweg: Das Array A enthalte alle Zahlen von 1 bis 1000. Anschließend werden alle Vielfachen der Zahlen 2, 3, 4, 5, …. gelöscht bzw. durch 0 überschrieben. Alle übrigen Zahlen sind Primzahlen.

11. Gegeben sei ein Zufallsfeld A und ein genauso großes, aber leeres Feld B (d.h. es enthält nur Nullen). Alle Zahlen aus A sollen folgendermaßen sortiert werden: Zunächst wird die kleinste Zahl aus A erstens in das Feld B an die erste Stelle B[1] geschrieben und zweitens durch die Konstante *Integer.MAX_VALUE* (im Feld A) überschrieben. Danach wird wieder die kleinste Zahl aus A ins Feld B an die Stelle B[2] geschrieben und im Feld A durch *Integer.MAX_VALUE* überschrieben, usw. Das Feld B entspricht anschließend dem sortierten Feld A. Das Feld B wird ausgegeben.

12. Gib alle Vornamen deiner Informatik-Kursmitglieder in ein Array ein. Der Computer soll das Feld sortieren und die Namen alphabetisch geordnet ausgeben.

Lösungen

Die folgende Methode wird bei fast jeder Aufgabe benötigt:

```
private void FeldausgabeA()    {
  taFeldA.setText("");
  for (int i=0; i < A.length; i++)
                              taFeldA.append(A[i]+"\n");
}


private void btAufgabe1MouseClicked(java….)  {
 A = new int [20];
 for (int i=0; i < A.length; i++)
                      A[i] = (int) (100*Math.random());
 FeldausgabeA();
}


private void btAufgabe2MouseClicked(java….)  {
  A = new int [100];
  for (int i=0; i < A.length; i++) A[i] = (i+1)*(i+1);
  FeldausgabeA();
}


private void btAufgabe3MouseClicked(java….)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                     A[i] = (int) (1000*Math.random());
  FeldausgabeA();

  int min = Integer.MAX_VALUE ;
  int minIndex = 0;

  for (int i = 0; i < A.length; i++)
     if(A[i] < min)  {
       min = A[i];
       minIndex = i;
     }
  taAusgabe.setText("");
  taAusgabe.append("Minimum = "+min + "\n");
  taAusgabe.append("Minimum-Index = "+minIndex);
}
```

```
private void btAufgabe4MouseClicked(java….)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                    A[i] = (int) (100*Math.random());
  FeldausgabeA();

  int sum = 0;
  double durchschnitt;

  for (int i = 0; i < A.length; i++)  sum = sum + A[i];
  durchschnitt = sum * 1.0 / A.length;
  // Beachte die Multiplikation mit 1.0  !
  taAusgabe.setText("");
  taAusgabe.append("Summe = "+sum + "\n");
  taAusgabe.append("Durchschnitt = "+durchschnitt);
}
```

```
private void btAufgabe5MouseClicked(java….)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                    A[i] = (int) (100*Math.random());
  FeldausgabeA();

  taAusgabe.setText("");
  for (int i=0; i < A.length; i++)
    if (A[i] % 2 == 0) taAusgabe.append(A[i] + "\n");
}
```

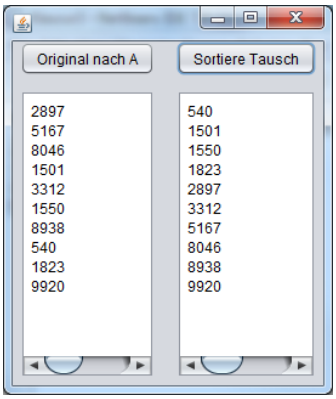
```
private void btAufgabe6MouseClicked(java….)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                    A[i] = (int) (100*Math.random());
  FeldausgabeA();

  String eingabe;
  eingabe = JOptionPane.showInputDialog(this, "Welche
                      Zahl möchtest du überprüfen?");
  int zahl = Integer.parseInt(eingabe);

  boolean enthalten = false;
```

```
  for (int i=0; i < A.length; i++)
     if (A[i] == zahl) enthalten = true;

  String meldung;
  if (enthalten) meldung = zahl + " ist im Feld
                                              enthalten";
  else meldung=zahl + " ist nicht im Feld enthalten";
  JOptionPane.showMessageDialog(this, meldung);
}
```

```
private void btAufgabe8MouseClicked(java…)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                    A[i] = (int) (100*Math.random());
  FeldausgabeA();

  for (int i=0; i < A.length; i++) A[i] = A[i]+1;
  for (int i=0; i < A.length; i++)
                          taAusgabe.append(A[i]+"\n");
}
```

```
private void btAufgabe9MouseClicked(java…)  {
  A = new int [20];
  for (int i=0; i < A.length; i++)
                    A[i] = (int) (100*Math.random());
  FeldausgabeA();

  int hilf = A[19];
  for (int i = 19; i > 0; i--)  A[i] = A[i-1];
  A[0] = hilf;

  for (int i=0; i < A.length; i++)
                          taAusgabe.append(A[i]+"\n");
}
```

Das folgende Programm zeigt einen weiteren Sortieralgorithmus:




```
public class Sortieren extends javax.swing.JFrame {

  private static final int n = 10;
  int[] Original = new int[n];
  int[] A = new int[n];

public Sortieren() {
  initComponents();
  for (int i = 0; i < n; i++ )
          Original[i] = (int) (Math.random()*10000);
  for (int i = 0; i < n; i++ )  {
    A[i] = Original[i];
    taAusgabe.append(A[i]+"\n");
  }
}


private void btOriginalNachAMouseClicked(java…) {
  for (int i = 0; i < n; i++ ) A[i] = Original[i];
}


private void tausche(int[] feld, int index1, int
index2)  {
  int hilf = feld[index1];
  feld[index1] = feld[index2];
  feld[index2] = hilf;
}
```

```
private void btSortiereTauschMouseClicked(java……) {
  for (int k = 0; k < n-1; k++)  {
     for (int i = k+1; i < n; i++)
        if (A[i] < A[k]) tausche(A, i, k);
  }
  for (int i = 0; i < n; i++ )
     taAusgabe2.append(A[i]+"\n");
  }
}
```

Qicksort

```
private void sort(int links, int rechts)  {
  int up, down, pivot, hilf;
  pivot = A[(links + rechts)/2];
  up = links;
  down =rechts;
  do  {
    while(A[up] < pivot) up++;
    while (A[down] > pivot) down--;
    if (up <= down)  {
       hilf =  A[down];
       A[down] = A[up];
       A[up] = hilf;
       up++;
       down--;
    }
  }
  while (up <= down);    // Ende der do-while-Schleife
  if (links < down)  sort(links, down);
  if (up < rechts)  sort(up, rechts);
}


private void btQickSortMouseClicked(java…) {
  sort(0,n-1);
}
```

Übergabe von Arrays an Methoden

Bei der Übergabe von beliebigen Arrays an Methoden muss beachtet werden, dass alle Arrays in Java Verweistypen sind – auch Arrays von elementaren Datentypen. Veränderungen von Arrayelementen in der Methode haben deshalb immer Auswirkung auf das ursprüngliche Array, welches der Methode übergeben wurde. Das folgende Beispiel zeigt die Problematik:

```
private void quadriere(int [] A)  {
  for (int i=0; i< A.length; i++) A[i] = A[i] * A[i];
}

private void btStartMouseClicked(java…..) {
  int [] feld = {1, 2, 3, 4, 5};
  for (int i=0; i < feld.length; i++)
                    System.out.print(feld[i]+"   ");
  System.out.println();
  quadriere(feld);
  for (int i=0; i < feld.length; i++)
                    System.out.print(feld[i]+"   ");
}
```

Obiges Programm ergibt folgende Ausgabe:

```
1   2   3   4   5
1   4   9   16   25
```

Zweidimensionale Arrays

Ein zweidimensionales Feld kann man sich wie eine Tabelle vorstellen.

```
int[][] A = new int[3][4]
```

Obiges Array namens A besteht aus 3 Zeilen (nummeriert **von 0 bis 2**) und 4 Spalten (nummeriert **von 0 bis 3**).

Zuweisungen geschehen etwa folgendermaßen: `A[1][2] = 15;`

Nur bei der Deklaration eines Arrays kann auch sofort eine Initialisierung erfolgen, indem direkt die Werte für das Array in geschweiften Klammern angegeben werden. Folgendes ist ein Beispiel für ein Feld mit 3 Zeilen und 4 Spalten:

```
int[][] A = { {5,3,7,2}, {8,6,9,12}, {6,-5,-9,8} }
```

Die Dimensionen können ebenfalls mit dem Attribut *length* abgefragt werden:
Die Anzahl der Zeilen eines zweidimensionalen Feldes ***A*** erhält man durch den Ausdruck ***A.length***
Die Anzahl der Spalten eines zweidimensionalen Feldes ***A*** erhält man durch den Ausdruck ***A.zeile.length*** wobei *zeile* eine gültige Zeilennummer sein muss.

```
for (int i = 0; i < A.length; i++)  {
   for (int j = 0; j < A[0].length; j++)
            System.out.print(A[i][j] + "   ");
   System.out.println();
}
```

Aufgaben mit zweidimensionalen Arrays

Benutze das Feld `int[][] A = new int[3][3]`.
Zunächst soll es natürliche Zufallszahlen zwischen 0 und 10 000 enthalten.

1. Gib das Feld (zweidimensional) in einer Textarea aus! Sorge dafür, dass alle Zahlen „4-stellig“ ausgegeben werden. Bei kleineren Zahlen sollen deshalb entsprechend viele Leerzeichen vor die Zahlausgabe gesetzt werden. Wähle für die Ausgabe in der Textarea die Schriftart *„Courier New“*, weil dann alle Zeichen (auch Leerzeichen) gleich breit dargestellt werden.

2. Gib die größte Zahl aus!

3. Gib die Summe aller Zahlen aus!

4. Gib die größte der 3 Zeilensummen aus!

5. Gib die größte der 3 Spaltensummen aus!

6. Gib die Summe der Hauptdiagonalen aus!

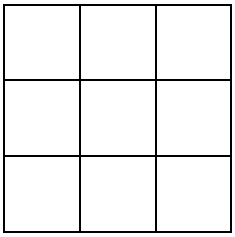
7. Gib folgendes Feld ein:

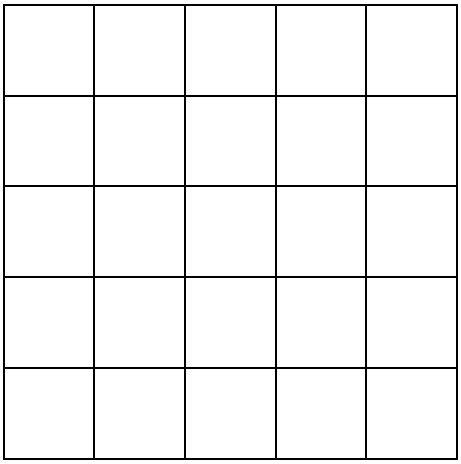
8	**1**	**6**
3	**5**	**7**
4	**9**	**2**

   Das Programm soll überprüfen, ob es sich um ein magisches Quadrat handelt!

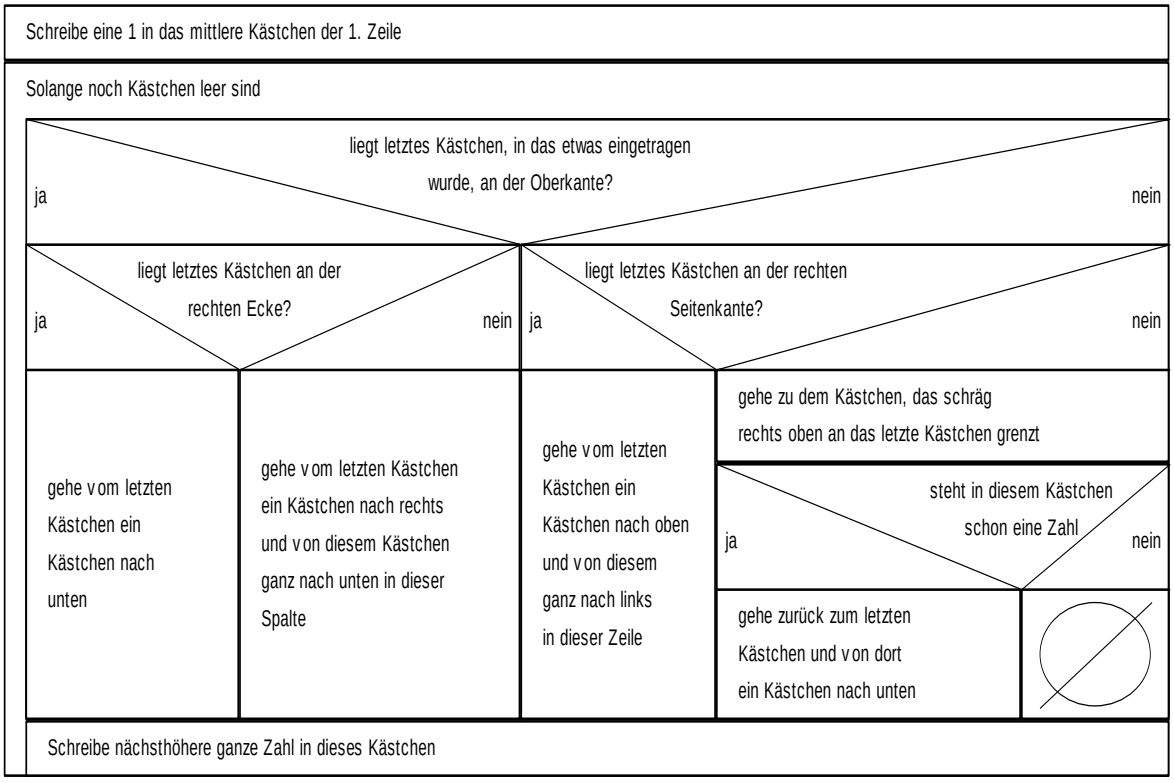
8. (Schwierig!) Erzeuge alle möglichen magischen 3×3-Quadrate und gib sie in einer Textarea aus! Die magische Summe beträgt 15.

9. Das folgende Struktogramm funktioniert nur für Zauberquadrate mit ungerader Zeilenanzahl (warum?).

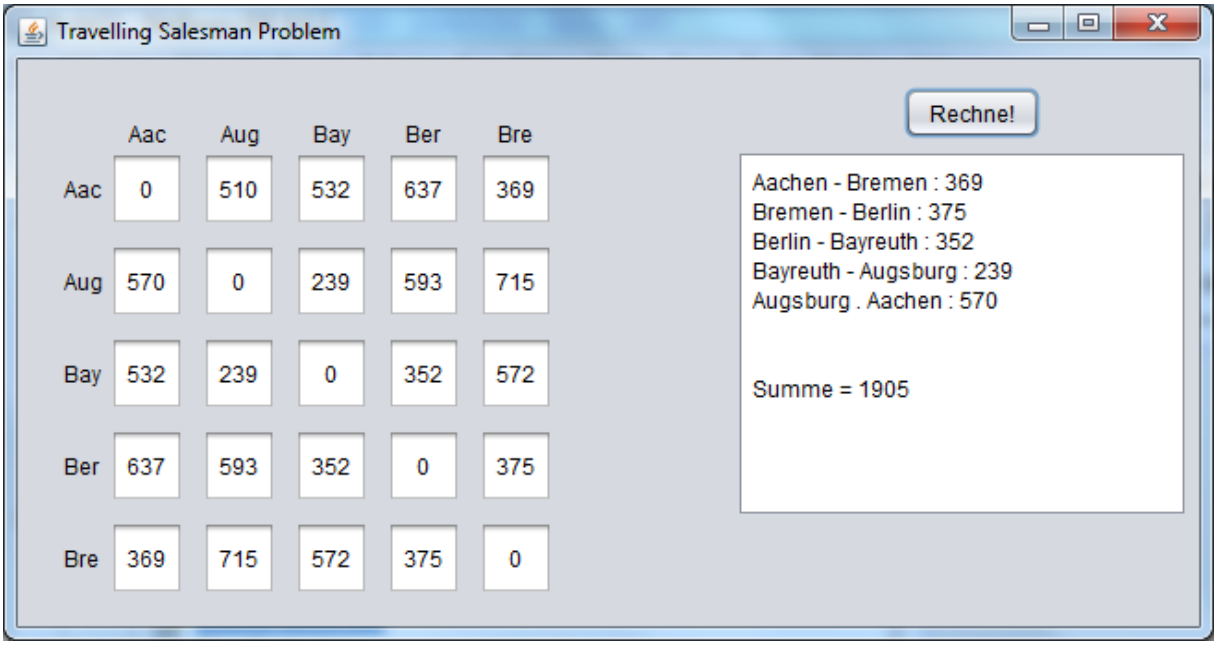




Magisches Quadrat



10. Erstelle nun ein Feld **`int[][] A = new int[5][5]`** , welches als eine Entfernungstabelle der fünf Orte ‘Ort1‘ bis ‘Ort5‘ angesehen werden kann. Diese Tabelle muss (natürlich?) symmetrisch sein und sie enthält in der Hauptdiagonalen nur Nullen.



a) Das Programm soll überprüfen, ob das Feld A symmetrisch ist.

b) Das Programm soll die Länge der folgenden Rundreise berechnen: Ort1-Ort2-Ort3-Ort4-Ort5-Ort1 !

c) Ermittle die kürzeste Rundreise mit der sog. *Next-Neigbour-Methode*: Man startet bei Ort1, fährt zum nächstgelegenen Nachbarort, von dort aus weiter zum nächsten noch nicht besuchten Ort usw.

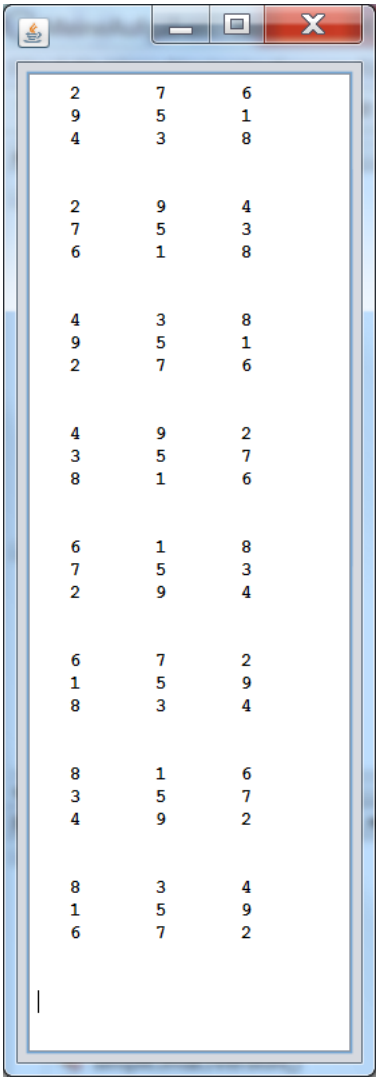
Lösungen

Aufgabe 1

```
private void feldausgabe()  {
  String zahlText;
  for (int i = 0; i<A.length; i++)   {
    zahlText = "";
    for (int j=0; j < A[0].length; j++) {
      zahlText = "" + A[i][j];
      if (A[i][j] < 1000) zahlText = " " + zahlText;
      if (A[i][j] < 100) zahlText = " " + zahlText;
      if (A[i][j] < 10) zahlText = " " + zahlText;
      ta.append(zahlText + "    ");
    }   //end of for j
    ta.append("\n");
  }   //end of for i

  ta.append("\n\n");
}
```

Aufgabe 8 (magische 3*3-Quadrate)

Es gibt 8 unterschiedliche magische 3*3-Quadrate (siehe Abbildung).



Man kann diese 3*3-Quadrate mit einem relativ einfachen, aber umständlichen und langen Programm erhalten (siehe unten).

Bemerkungen dazu:
Das unten stehende Programm wird (im Jahre 2018) „blitzschnell“ durchgeführt. Nach dem Starten sind praktisch sofort alle 8 Lösungen vorhanden.

Man kann untenstehendes Programm ziemlich einfach derart umschreiben, dass es alle magischen 4*4-Quadrate erzeugt.

Eine leichte mathematische Überlegung zeigt, dass die Berechnung aller magischen 4*4-Quadrate fast 60 Millionen Mal länger dauern wird als diese Berechnung aller 3*3-Quadrate.

Sollte untenstehendes Programm also (im Jahre 2018) etwa 1 ms benötigen, so bräuchte man für alle 4*4-Quadrate fast einen ganzen Tag.

Erstens lässt sich unten stehendes Programm durchaus noch sehr beschleunigen (man sollte z.B. nicht erst nach dem Setzen der letzten Zahl das Quadrat überprüfen, sondern schon wesentlich eher und häufiger!).

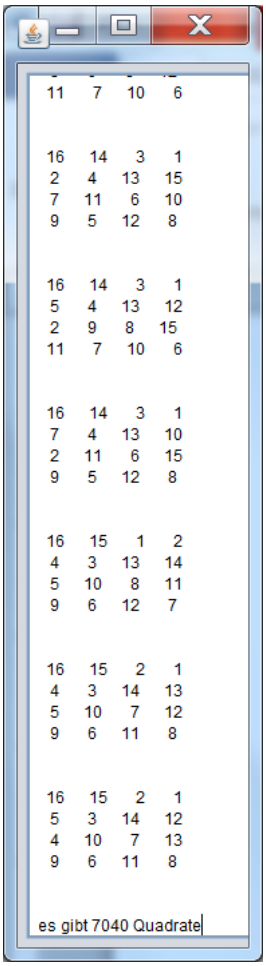
Zweitens gibt es einen wesentlich besseren Algorithmus für das Erzeugen aller n*n-Quadrate (Stichwort: Backtracking).

```
public class MagicSquare extends javax.swing.JFrame {

   int n = 3;
   int magicSum = (1+n*n)*n/2;
   int[] [] A = new int[n] [n];
   Boolean belegt[] = new Boolean[n*n+1];
   Boolean esHatGeklappt;

   public MagicSquare() {
     initComponents();
     for (int i=1; i<=n*n; i++) belegt[i]=false;
     esHatGeklappt = false;
     ta.setText("");
     simple3mal3Version();
   }


private Boolean pruefeQuadrat() {
  Boolean ergebnis = true;
  if (A[0][0]+A[0][1]+A[0][2] != magicSum) ergebnis=false;
  else if (A[1][0]+A[1][1]+A[1][2] != magicSum) ergebnis=false;
       else if (A[2][0]+A[2][1]+A[2][2] != magicSum)
               ergebnis=false;

  else if (A[0][0]+A[1][0]+A[2][0] != magicSum) ergebnis=false;
       else if (A[0][1]+A[1][1]+A[2][1] != magicSum)
               ergebnis=false;
            else if (A[0][2]+A[1][2]+A[2][2] != magicSum)
                    ergebnis=false;

  else if (A[0][0]+A[1][1]+A[2][2] != magicSum) ergebnis=false;
       else if (A[0][2]+A[1][1]+A[2][0] != magicSum)
              ergebnis=false;

  return ergebnis;
}


private void simple3mal3Version() {
  for (int a=1; a<=9; a++) {
    belegt[a]=true;
    A[0][0]=a;
    for (int b=1; b<=9; b++) {
      if (!belegt[b]) {
        belegt[b]=true;
```

```
A[0][1]=b;

for (int c=1; c<=9; c++) {
  if (!belegt[c]) {
    belegt[c]=true;
    A[0][2]=c;

    for (int d=1; d<=9; d++) {
      if (!belegt[d]) {
        belegt[d]=true;
        A[1][0]=d;

        for (int e=1; e<=9; e++) {
          if (!belegt[e]) {
            belegt[e]=true;
            A[1][1]=e;

            for (int f=1; f<=9; f++) {
              if (!belegt[f]) {
                belegt[f]=true;
                A[1][2]=f;

                for (int g=1; g<=9; g++) {
                  if (!belegt[g]) {
                    belegt[g]=true;
                    A[2][0]=g;

                    for (int h=1; h<=9; h++) {
                      if (!belegt[h]) {
                        belegt[h]=true;
                        A[2][1]=h;

                        for (int i=1; i<=9; i++) {
                          if (!belegt[i]) {
                            belegt[i]=true;
                            A[2][2]=i;

                            esHatGeklappt =
                                      pruefeQuadrat();
                            if (esHatGeklappt)
                                        feldausgabe();

                                  belegt[i]=false;
                        } // of if i
                      // of for i
```

```
                belegt[h]=false;
              } // of if h
            } // of for h


            belegt[g]=false;
          } // of if g
        } // of for g


          belegt[f]=false;
        } // of if f
      } // of for f


          belegt[e]=false;
        } // of if e
      } // of for e


          belegt[d]=false;
        } // of if d
      } // of for d

          belegt[c]=false;
        } // of if c
      } // of for c


        belegt[b]=false;
      } // of if b
    }
    belegt[a]=false;
  } //of for a
}
```

Aufgabe 8 (magische n*n-Quadrate, Backtracking)



Es ist offensichtlich, dass durch Rotation um 90°, 180° und 270° sowie durch Spiegelung an den Hauptachsen und Diagonalen aus einem magischen Quadrat wieder ein magisches Quadrat entsteht. Diese acht magischen Quadrate sind äquivalent; es genügt, eines davon zu untersuchen. Deswegen gibt es eigentlich auch nur 7040 : 8 = 880 wirklich unterschiedliche magische Quadrate mit Kantenlänge 4. Alle diese 880 Quadrate wurden bereits 1693 gefunden.
Es gibt ein (triviales) magisches Quadrat mit Kantenlänge 1, jedoch keines mit Kantenlänge 2. Abgesehen von Symmetrieoperationen gibt es nur 1 magisches Quadrat der Kantenlänge 3. Mit Kantenlänge 5 gibt es 275.305.224 magische Quadrate; darüber hinaus sind keine genauen Zahlen bekannt.

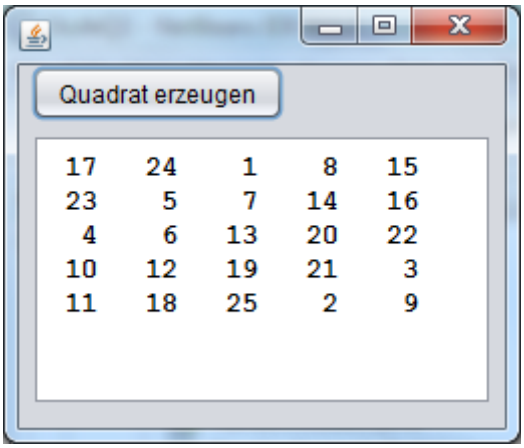
Das nachfolgende Programm zur Berechnung aller 4*4-Zauberquadrate benötigte im Jahre 2018 auf einem 64-Bit-Rechner etwa 3 Minuten.

```
public class MagicSquareBacktracking extends …..{

  int n = 4;
  int magicSum = (1+n*n)*n/2;
  int[][] A = new int[n][n];
  Boolean belegt[] = new Boolean[n*n+1];
  Boolean esGehtNoch;
  int counter = 0;


  public MagicSquareBacktracking() {
    initComponents();
    for (int i=1; i<=n*n; i++) belegt[i]=false;
    esGehtNoch = false;
    ta.setText("");
    setzeNummer(1);
    ta.append("es gibt " + counter + " Quadrate");
  }
```

```
private void feldausgabe()  {
  String zahlText;
  for (int i = 0; i<A.length; i++)   {
     zahlText = "";
     for (int j=0; j < A[0].length; j++) {
        zahlText = "" + A[i][j];
        if (A[i][j]<1000) zahlText = " " + zahlText;
        if (A[i][j]<100) zahlText = " " + zahlText;
        if (A[i][j]<10) zahlText = " " + zahlText;
        ta.append(zahlText + "    ");
     }
     ta.append("\n");
  }
  ta.append("\n\n");
}


private Boolean pruefeQuadrat(int zeile, int spalte) {
  int zeilensumme = 0;
  int spaltensumme = 0;
  int diagonale1 = 0;
  int diagonale2 = 0;
  Boolean ergebnis = true;

if (spalte == n-1) {
  for (int i = 0; i<n; i++) zeilensumme = zeilensumme + A[zeile][i];
  ergebnis = (zeilensumme == magicSum);
}

if (ergebnis && zeile == n-1) {
   for (int j=0; j<n; j++) spaltensumme = spaltensumme+A[j][spalte];
   ergebnis = (spaltensumme == magicSum);
}

if (ergebnis && zeile == n-1 && spalte == 0) {
   for (int k = 0; k<n; k++) diagonale2 = diagonale2 + A[k][n-1-k];
   ergebnis = (diagonale2 == magicSum);
}

if (ergebnis && zeile == n-1 && spalte == n-1) {
   for (int l = 0; l<n; l++) diagonale1 = diagonale1 + A[l][l];
   ergebnis = (diagonale1 == magicSum);
}
return ergebnis;
}
```

```
private void setzeNummer(int nummer) {
   int zeile = (nummer - 1) / n;
   int spalte = (nummer - 1) % n;

   for (int zahl=1; zahl<=n*n; zahl++) {
   if (!belegt[zahl])  {
      A[zeile][spalte]=zahl;
      esGehtNoch = pruefeQuadrat(zeile, spalte);
      if (esGehtNoch) {
        if (nummer == n*n) {
           feldausgabe();
           counter++;
        }
        else {
          belegt[zahl]=true;
          setzeNummer(nummer + 1);

          belegt[zahl] = false;
        }
      } //end of if esGehtNoch

   } //end of if !belegt[zahl]
  } // end of for zahl

}
```

Aufgabe 9 (magisches Quadrat)




```
int n=5;
int [] [] A = new int[n] [n];

public MagischesQuadrat() {
     initComponents();
     for (int i=0; i<n;i++)
          for (int j=0; j<n; j++) A[i] [j] = 0;
}


private void feldausgabe() {
 //siehe Lösung der Aufgabe 1!


private void btErzeugenMouseClicked(java.awt…..) {
  int nummer = 1;
  int zeile = 0;
  int spalte = n/2;
  A[zeile][spalte] = nummer;
  while (nummer < n*n)   // solange noch Kästchen leer sind
  {
    if (zeile == 0)     // liegt letztes Kästchen oben?
    {
      if (spalte == n-1) // liegt letztes Kästchen an der rechten Ecke?
      {
        zeile = zeile + 1;
      }
      else {
         spalte = spalte + 1;
         zeile = n-1;
      }
    }
```

```
    else // letztes Kästchen liegt nicht oben
    {
      if (spalte == n-1)   // liegt letztes Kästchen an der rechten Seitenkante?
      {
        zeile = zeile - 1;
        spalte = 0;
      }
      else {
        zeile = zeile - 1;
        spalte = spalte + 1;
        if (A[zeile] [spalte] != 0) //steht da schon eine Zahl?
        {
          zeile = zeile + 1;
          spalte = spalte - 1;
          zeile = zeile + 1;
        }
      }
    }

    nummer = nummer + 1;
    A[zeile][spalte] = nummer;
  } // Ende von while

  feldausgabe();
}
```

Aufgabe 10 (Rundreise)

```
import javax.swing.*;   // für JTextField
public class Handlungsreise extends javax.swing.JFrame {
  int n = 5;
  int [][] A = { {0,510,532,637,369}, {570,0,239,593,715},
                 {532,239,0,352,572}, {637,593,352,0,375},
                 {369,715,572,375,0} };
  JTextField [] [] TF = new JTextField [n][n];
  String [] Ort = {"Aachen", "Augsburg", "Bayreuth", "Berlin",
                   "Bremen"};
  JLabel []  LabZeile = new JLabel[n];
  JLabel []  LabSpalte = new JLabel[n];
  Boolean [] besetzt = new Boolean[n];

  public Handlungsreise() {
    initComponents();

    TF = new JTextField [n] [n];
    for (int i=0; i<n; i++) {
      LabZeile[i] = new JLabel();
      LabZeile[i].setText(Ort[i].substring(0,3));
      LabZeile[i].setBounds(25, 50+50*i, 40, 40);
      add(LabZeile[i]);

      LabSpalte[i] = new JLabel();
      LabSpalte[i].setText(Ort[i].substring(0,3));
      LabSpalte[i].setBounds(60+50*i, 20, 40, 40);
      add(LabSpalte[i]);
   }

   for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)  {
       TF[i][j] = new JTextField();
       TF[i][j].setBounds(50+50*j, 50+50*i, 40, 40);
       //Beachte Koordinaten!
       TF[i][j].setHorizontalAlignment(JTextField.CENTER);
       TF[i][j].setText(""+A[i][j]);
       add(TF[i][j]);
     }

   for (int i=0; i<n; i++) besetzt[i] = false;
  }
```

```
private int naechsterOrt(int start) {
  int min = 40000;
  int index=0;
  int abstand;
  for (int lauf = 0; lauf < n; lauf++)
    if (lauf != start && besetzt[lauf] == false)  {
       abstand = A[start][lauf];
       if (abstand < min) {
          min = abstand;
          index = lauf;
       }
    }

  besetzt[index] = true;
  return index;
}


private void btRechneMouseClicked(java.awt.....)  {
  int summe = 0;
  int start = 0; // wilkürlicher Startort;
  int next = 0;   // Dummy-Zahl;
  int lauf = start;
  besetzt[start] = true;
  for (int i=1; i<n; i++)  {
     tfReise.append(Ort[lauf]+" - ");
     next = naechsterOrt(lauf);
     tfReise.append(Ort[next]+" : "+A[lauf][next]+"\n");
     summe = summe + A[lauf][next];
     lauf = next;
  }

  tfReise.append(Ort[lauf]+" . "+Ort[start]+" : "
                                          +A[lauf][start] +"\n");
  summe = summe + A[lauf][start];
  tfReise.append("\n\nSumme = "+summe);
}


}
```

Matrizenrechnungen

1. Für die Determinante einer Matrix $M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$ gilt:

det M = aei + bfg + cdh – ceg – bdi – afh
(Produkt der drei Hauptdiagonalen minus Produkt der drei Nebendiagonalen)

2. Für das Produkt einer Matrix mit einem Vektor gilt:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

3. Für das Produkt zweier Matrizen gilt:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \cdot \begin{pmatrix} x & u & r \\ y & v & s \\ z & w & t \end{pmatrix} = \begin{pmatrix} (ax + by + cz) & (au + bv + cw) & (ar + bs + ct) \\ (dx + ey + fz) & (du + ev + fw) & (dr + es + ft) \\ (gx + hy + iz) & (gu + hv + iw) & (gr + hs + it) \end{pmatrix}$$

4. Für das Produkt einer Matrix mit einer Zahl gilt:

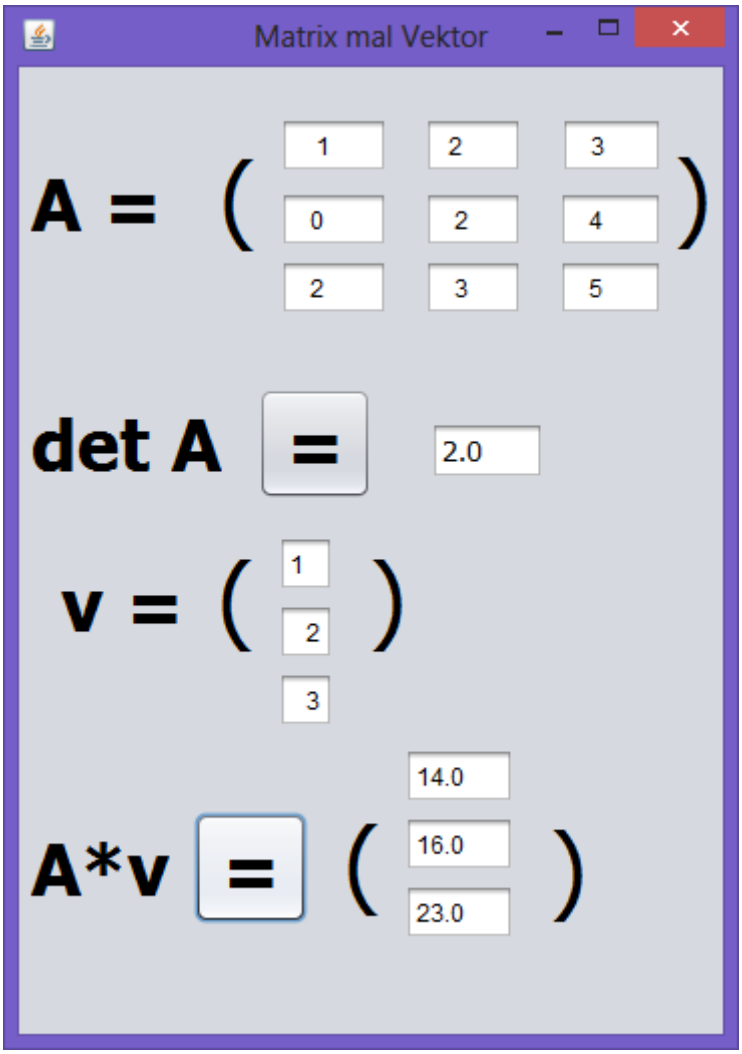
$$k \cdot \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} ka & kb & kc \\ kd & ke & kf \\ kg & kh & ki \end{pmatrix}$$

5. Für die Inverse einer 3x3-Matrix gilt:

$$A^{-1} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}^{-1} = \frac{1}{\det(A)} \begin{pmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{pmatrix}$$

Lösungen

Aufgaben 1 und 2




```
public class MRechnungen extends javax.swing.JFrame {

  double [] [] A = new double [3] [3];
  double [] v = new double [3];
  double [] e = new double [3];
  ……..

private void liesA()  {
  A[0][0]= Double.parseDouble(tf00.getText());
  A[0][1]= Double.parseDouble(tf01.getText());
  A[0][2]= Double.parseDouble(tf02.getText());
  A[1][0]= Double.parseDouble(tf10.getText());
```

```
  A[1][1]= Double.parseDouble(tf11.getText());
  A[1][2]= Double.parseDouble(tf12.getText());
  A[2][0]= Double.parseDouble(tf20.getText());
  A[2][1]= Double.parseDouble(tf21.getText());
  A[2][2]= Double.parseDouble(tf22.getText());
}


private void liesVektor() {
  v[0] = Double.parseDouble(tfx.getText());
  v[1] = Double.parseDouble(tfy.getText());
  v[2] = Double.parseDouble(tfz.getText());
}


private void btDetMouseClicked(java……) {
  liesA();
  double d = A[0][0]*A[1][1]*A[2][2]
    +A[0][1]*A[1][2]*A[2][0] +A[0][2]*A[1][0]*A[2][1]
    -A[0][0]*A[1][2]*A[2][1] -A[0][1]*A[1][0]*A[2][2]
    -A[0][2]*A[1][1]*A[2][0];
  tfDet.setText(""+d);

}


private void
btMatrixMalVektorMouseClicked(java.awt.event..) {
  liesA();
  liesVektor();
  for (int i=0; i<3; i++)
    e[i] = A[i][0]*v[0] +A[i][1]*v[1] + A[i][2]*v[2];
  tf1.setText("" + e[0]);
  tf2.setText("" + e[1]);
  tf3.setText("" + e[2]);
}
} //of class MRechnungen
```

Zeitmessungen

Die Methode *System.currentTimeMillis()* liefert (als Datentyp *long*) die Anzahl der Millisekunden, die seit dem 01.01.1970 vergangen sind (das sind - im Jahre 2019 - mittlerweile rund 1,5 Billionen). Mit dieser Methode lassen sich Zeiten (insbesondere natürlich auch Zeitdifferenzen) messen.

Leider liefert diese Methode bei Aufruf nicht den aktuellen Zeitstand. Sie aktualisiert sich nur etwa alle 1 ms (im Jahre 2019). Das hängt teilweise auch davon ab, wie viele Programme gleichzeitig laufen.
Beachte, dass unsere heutigen Rechner eine Taktfrequenz von mehreren GHz besitzen, d.h. ein Takt dauert weniger als eine Nanosekunde.
Innerhalb einer Millisekunde finden also mehr als 1 Million Takte statt, in denen Befehle ausgeführt werden. Ein einfacher Grundbefehl wie etwa die Addition zweier Zahlen dauert nur wenige Takte.

```
public long startZeit;

private void zeitAngabe(long t)  {
   int ms = (int) (t % 1000);
   t = t / 1000; // t = Anzahl Sekunden
   int s = (int) (t % 60);

   t = t / 60;   // t = Anzahl Minuten
   int m = (int) (t % 60);

   t = t / 60;   // t = Anzahl Stunden
   int st = (int) (t % 24);

   t = t / 24;   // t = Anzahl Tage
   int d = (int) (t % 365);

   int y = (int) (t / 365);   // ohne Berücksichtigung Schaltjahre
                              // y = Anzahl Jahre
   System.out.println(y+"y "+d+"d "+st+"h "+m+"m "
                                       +s+"s "+ms+"ms");
}
```

```
private void btStartMouseClicked(java…..) {
   startZeit = System.currentTimeMillis();
   System.out.println(startZeit);
   zeitAngabe(startZeit);
}
```

Die Ausgabe für obiges Programm lautet (am 26. Juli 2013; beachte, dass Schaltjahre bei der Umrechnung nicht berücksichtigt worden sind!):
1374829746304
43y 217d 9h 9m 6s 304ms

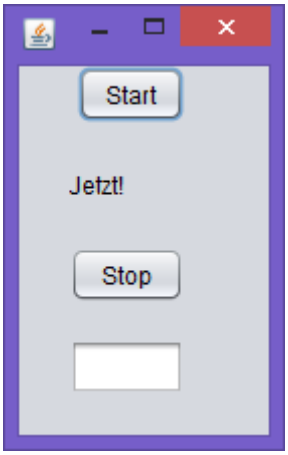
Aufgaben

1. Erstelle einen Start- und einen Stoppbutton, mit dem du Zeiten messen kannst. Die Zeitdifferenz soll in einem Textfeld ausgegeben werden. Wie gut kannst du 20 Sekunden abschätzen?

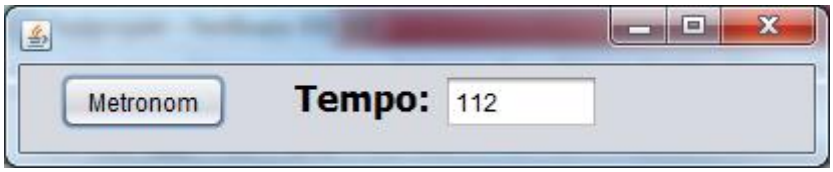
2. Schreibe eine Methode *public warte(long delta);* welche *delta* Millisekunden lang wartet.
Hinweis: setze `long t1 = System.currentTimeMillis();`
Frage dann solange immer wieder die Zeit t2 ab, bis t2 – t1 > delta.

3. Ermittle folgendermaßen die Genauigkeit der Zeitangabe obiger Methode *System.currentTimeMillis()* :
Speichere das Ergebnis dieses Aufrufes in einer Variablen t1. Frage danach immer wieder diese Methode ab und speichere das Ergebnis in der Variablen t2, solange bis t2 ungleich t1. Gib die Differenz t2 – t1 aus. Wiederhole diese Messung 100 Mal und ermittle den Mittelwert!

4. Schreibe ein Programm zur Messung der Reaktionszeit. Nach Betätigung eines Buttons Start wird eine zufällige Zeit lang gewartet und anschließend das Wort „Jetzt!“ ausgegeben. Nach Betätigung eines Stop-Buttons wird die benötigte Zeit ausgegeben.
System.currentTimeMillis() :



5. Simuliere ein Metronom! Ein Metronom gibt die Anzahl der Viertelnoten pro Minute an. Im folgenden Programm wird im Takt der Musik (Viertelnoten!) mit der Maus auf den Button Metronom geklickt. Das Tempo wird errechnet und ausgegeben.



Mache eine mathematische Aussage zur Genauigkeit dieses Metronoms! Angenommen, das Metronom zeigt das Tempo 120 an. Welchen Einfluss hat das Aktualisierungsintervall von 1ms der internen Uhr auf die Tempo-angabe?

6. Sortiere ein Integerfeld mit n = 5 000 zufälligen Zahlen zwischen 0 und 100 000. Untersuche, wie die benötigte Zeit von der Anzahl n abhängt!

7. Vergleiche mehrere Sortieralgorithmen bezüglich der benötigten Zeit! Beachte, dass alle Algorithmen dasselbe Zufallsfeld sortieren. Untersuche auch schon unterschiedlich vorsortierte Zahlenfelder (vorwärts oder rückwärts sortiert)!

Lösungen

Aufgabe 2:

```
private void warte(long delta)  {
   long t1 = System.currentTimeMillis();
   long t2 = System.currentTimeMillis();
   while (t2-t1 < delta)
      t2 = System.currentTimeMillis();
}
```

Aufgabe 3:

```
private void btStartMouseClicked(java….) {
  taAusgabe.setText("");
  long t1;
  long t2;
  long summe = 0;
  for (int i = 0; i<100; i++)  {
     t1 = System.currentTimeMillis();
     t2 = System.currentTimeMillis();
     while (t2-t1 == 0) t2=System.currentTimeMillis();

     summe = summe + (t2-t1);
     taAusgabe.append((t2-t1)+"\n");
  }
  tfDurchschnitt.setText(""+summe/100);
}
```

Aufgabe 4:

```
long start;
long stop;

private void btStartMouseClicked(java…..) {
   tfAusgabe.setText("   ");
   warte((long) (Math.random()*4000)); //vgl. Aufgabe 2
   start = System.currentTimeMillis();
   lab1.setText("Jetzt!");
}

private void btStopMouseClicked(java…..) {
   stop = System.currentTimeMillis();
   lab1.setText("");
   int delta = (int) (stop - start);
   tfAusgabe.setText("" + delta);
}
```

Aufgabe 5:

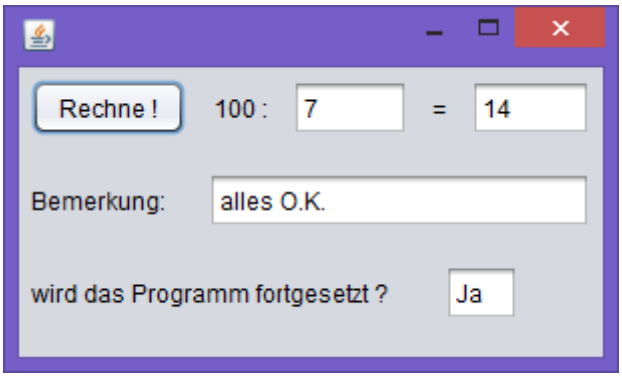
```
long startZeit;
long endZeit;
long delta;
long tempo;
int count = 0;

private void btMetronomMouseClicked(java…) {
  count++;
  if (count == 1)
     startZeit = System.currentTimeMillis();
  else  {
    endZeit = System.currentTimeMillis();
    delta = endZeit - startZeit;
    startZeit = endZeit;
    tempo = 60000/delta;
    tfAusgabe.setText("" + tempo);
  }
}
```

Exceptions-Fehlerbehandlung

Es gibt eine Reihe von möglichen Fehlern, die während der Laufzeit eines Programmes auftreten, und die das System Java selbst feststellt und dann mitteilt. Zu diesen Fehlern gehören insbesondere:
Division durch Null, unzulässiger Index bei Arrays, Eingabe von nicht erwarteten Zeichen über die Tastatur (z.B. ein Buchstabe bei erwarteter Integer-Eingabe), Zugriffsversuch auf eine nicht existierende Datei bzw. unbekannte Pfadangabe.

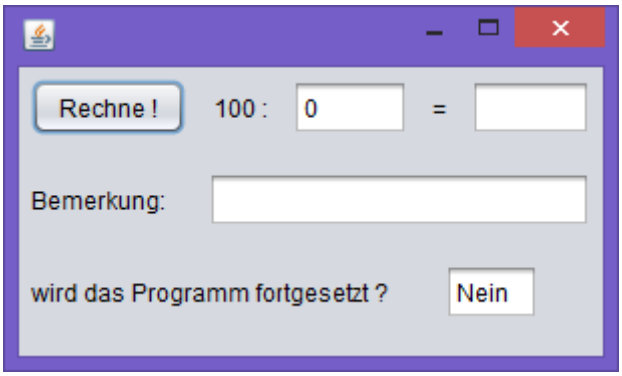
Damit das Programm nicht abstürzt, sollten Programmteile, in denen Laufzeitfehler auftreten könnten, in einem sog. *try and catch* Block ausgeführt werden. Im *try*-Teil stehen die kritischen Anweisungen, im *catch*-Teil stehen die Anweisungen für den Fall, dass ein Fehler aufgetreten sein sollte.

Beispiel: Im folgenden Programm soll die Zahl 100 durch eine im Textfeld *tfEingabe* eingegebene Zahl dividiert werden. Das Ergebnis soll im Textfeld *tfAusgabe* ausgegeben werden. Bei korrekter Eingabe sieht das Programm so aus:




```
private void btRechneMouseClicked(java…..) {
    tfFortsetzung.setText("Nein");
    int n = Integer.parseInt( tfEingabe.getText());
    int ergebnis = 100 / n;
    tfAusgabe.setText(""+ergebnis);
    if (n==0) tfBemerkung.setText("Div durch Null");
    else tfBemerkung.setText("alles O.K.");
    tfFortsetzung.setText("Ja");
}
```

Bei Eingabe der zum Fehler führenden Zahl Null erhält man folgendes Ergebnis:



Zusätzlich erscheinen im System-Output-Fenster jede Menge Fehlermeldungen.
Wichtig sind zwei Dinge:

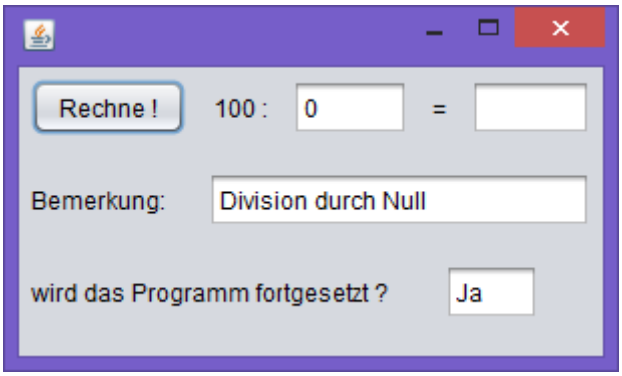
1. Der Rechner stürzt nicht vollständig ab. Stattdessen kann man das Programm durchaus weiterhin benutzen, also eine neue Zahl eingeben und neu rechnen lassen.
2. Nach Auftreten des Fehlers unterbricht das Programm genau an der Stelle, an welcher der Fehler entstand. Weitere Anweisungen werden nicht mehr ausgeführt!

Zur Erklärung:
Das Programm hat eine Fehlermeldung vom Typ *Exception* auf dem System-Output-Fenster ausgegeben.

Diese System-Fehlerausgabe lässt sich mit *try and catch* abfangen:

```
private void btRechneMouseClicked(java…..) {
    tfFortsetzung.setText("Nein");
    try  {
      int n = Integer.parseInt( tfEingabe.getText());
      int ergebnis = 100 / n;
      tfAusgabe.setText(""+ergebnis);
      tfBemerkung.setText("alles O.K.");
    }
    catch(Exception e)  {
       tfBemerkung.setText("Division durch Null");
    }
    tfFortsetzung.setText("Ja");
}
```

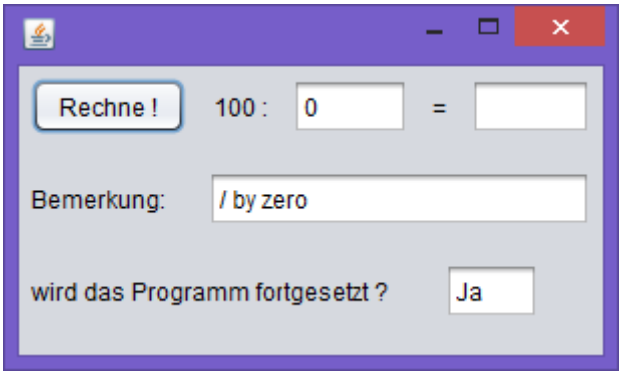
Obige neue Programmversion liefert nun bei Eingabe der Zahl Null folgendes Ergebnis:



Außerdem gibt es jetzt auch keine Fehlermeldungen im Konsolen-Fenster mehr.
Eine äquivalente *catch*-Programmierung wäre:

```
catch(Exception e)   {
    tfBemerkung.setText(e.getMessage());
}
```

Dies führt zu folgender Meldung:



In der obigen *catch*-Klausel wird ein formaler Parameter e angegeben, der beim Auftreten der Ausnahme ein Fehlerobjekt übernehmen soll. Fehlerobjekte sind dabei Instanzen der Klasse *Throwable* oder einer ihrer Unterklassen. Sie werden vom Aufrufer der Ausnahme erzeugt und als Parameter an die *catch*-Klausel übergeben. Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers. So liefern beispielsweise die Methoden *getMessage()* oder *toString()* einen Fehlertext (wenn dieser explizit gesetzt wurde).

Die Basisklasse für alle Ausnahmefehler heißt *Throwable*. Zwei direkte Unterklassen davon sind die beiden Klassen *Error* und *Exception*.
Die Klasse *Error* ist Superklasse aller *schwerwiegenden* Fehler. Diese werden hauptsächlich durch Probleme in der virtuellen Java-Maschine ausgelöst und

können auf Schulniveau nicht gelöst werden.

Alle Fehler, die möglicherweise für die Anwendung selbst von Interesse sind, befinden sich in der Klasse *Exception* oder einer ihrer Unterklassen. Ein Fehler dieser Art signalisiert einen abnormen Zustand, der vom Programm abgefangen und behandelt werden kann.

Ein catch-Block mit einem Parameter der Klasse *Exception* fängt jeden Fehler (außer *Error*-Ausnahmen) ab. Damit ist das Abfangen noch sehr unspezifisch, weil es viele unterschiedliche *Exceptions* gibt. Alternativ kann man mehrere hintereinander folgende *catch*-Blöcke schreiben, die dann teilweise auch speziellere *Exception*-Unterklassen-Parameter haben:

```
catch(NumberFormatException e)  {
   tfBemerkung.setText(e.toString());
}
catch(Exception e)  {
   tfBemerkung.setText("anderer Fehler:"+e.getMessage());
}
```

Die Klasse *IOException* ist eine direkte Unterklasse von *Exception*. Die Klasse *IOException* hat weitere Unterklassen, beispielsweise *NumberFormatException*, *FileNotFoundException*, *IndexOutOfBoundsException*, *NullPointerException*.

Der *finally*-Block

Wenn nach einem *try*-Block gewisse Anweisungen unbedingt ausgeführt werden müssen, so kann ein sogenannter *finally*-Block nach den *catch*-Blöcken implementiert werden. Dieser Block wird immer ausgeführt, egal ob eine Exception ausgelöst wurde oder nicht.
(Allerdings wird das Programm sowieso fortgesetzt. Welchen Nutzen hat also dieser finally-Block? Man könnte dessen Anweisungen doch einfach als natürliche Fortsetzung im Programm schreiben. Dann würden diese Anweisungen auch auf jeden Fall ausgeführt – falls eine *catch*-Anweisung vorhanden ist).

Ausnahmen werfen

Alle bisherigen Ausnahmen wurden von der Java-Umgebung automatisch „geworfen“. Es ist jedoch auch möglich, dass der Programmierer selbst eine Ausnahme mithilfe von *throw* wirft. Es muss nur eine Instanz einer Exception-Klasse angegeben werden.

```
.....
try  {
   int n = Integer.parseInt( tfEingabe.getText());
   int ergebnis = 100 / n;
   tfAusgabe.setText(""+ergebnis);
   tfBemerkung.setText("alles O.K.");
   if (n == 5) throw (new ArithmeticException());
}

catch(ArithmeticException ae)  {
   System.out.println("ich will nicht durch 5 teilen");
}
.....
```

Zeichnen

Wichtig: für das Folgende muss das Paket *java.awt. importiert werden!**

Man kann auf jedem Element der GUI auch frei selber zeichnen (also Punkte, Linie, Rechtecke, Kreise etc). Das Verfahren ist immer dasselbe. Jede Komponente einer GUI (*JFrame, JPanel, JButton*) hat einen „Zeichenagenten". Den muss man holen und in einer Variablen vom Typ *Graphics* speichern, Mit Hilfe dieser Variablen kann man ihn dann beauftragen, Dinge auf die Komponente zu zeichnen. Angenommen, man möchte auf einem *JPanel* zeichnen, das den Variablennamen *panZeichenbrett* hat. Dann holt man sich zuerst den Zeichenagenten:

```
Graphics zeichenagent =panZeichenbrett.getGraphics();
zeichenagent.drawRect(10,20,50,60);
zeichenagent.fillOval(80,90,100,110);
```

Manchmal möchte man auf dem Formblatt selbst zeichnen und der Zeichenagent sollte ein globales Objekt sein. Dann deklariert man den Zeichenagenten global und initialisiert ihn in der Konstruktormethode:

```
import java.awt.*;
public class Klausur extends javax.swing.JFrame {

  Graphics zeichenagent;

  public Klausur() {
    initComponents();
    zeichenagent = this.getGraphics();
  }

  private void …….
```

Die wichtigsten Operationen, die der Zeichenagent ausführen kann, sind *drawRect(x-Position, y-Position, breite, hoehe), drawOval(x-Position, y-Position, breite, hoehe), drawPolygon(int[] arx, int[] ary, int cnt), drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)* und *drawLine(int x1, int y1, int x2, int y2).*

Außerdem gibt es die ersten vier Methoden noch als *fill....* für gefüllte Formen. Dabei beziehen sich die Koordinatenangaben immer auf die obere linke Ecke der Figur (auch bei Ovalen!).

Die beiden Polygon-Methoden erwarten drei Parameter. Der erste ist ein Array mit einer Liste der *x*-Koordinaten und der zweite ein Array mit einer Liste der *y*-Koordinaten. Beide Arrays müssen so synchronisiert sein, dass ein Paar von Werten an derselben Indexposition immer auch ein Koordinatenpaar ergibt. Die Anzahl der gültigen Koordinatenpaare wird durch den dritten Parameter festgelegt. Das Polygon wird nach Abschluss der Ausgabe automatisch geschlossen. Falls der erste und der letzte Punkt nicht identisch sind, werden diese durch eine zusätzliche Linie miteinander verbunden.

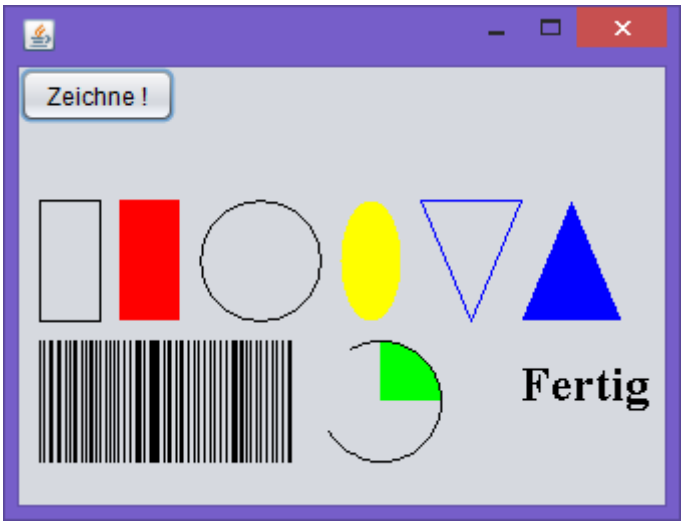
Folgendermaßen kann man einen Punkt setzen:
```
zeichenagent.drawLine(x,y,x,y);
```

Ein Kreisbogen ist ein zusammenhängender Abschnitt der Umfangslinie eines Kreises. Er kann mit folgender Methode gezeichnet werden:
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
Die ersten vier Parameter bezeichnen dabei den Kreis bzw. die Ellipse so, wie dies auch bei `drawOval` der Fall war. Mit `startAngle` wird der Winkel angegeben, an dem mit dem Kreisabschnitt begonnen werden soll, und *arcAngle* gibt den zu überdeckenden Bereich an. Dabei bezeichnet ein Winkel von 0 Grad die 3-Uhr-Position, und positive Winkel werden entgegen dem Uhrzeigersinn gemessen. Als Einheit wird *Grad* verwendet und nicht das sonst übliche Bogenmaß.

Analog kann man einen gefüllten Kreisausschnitt zeichnen:
`fill`*Arc(int x, int y, int width, int height, int startAngle, int arcAngle)*

Die einfachste Methode, Text auszugeben, besteht darin, *drawString(String str, int x, int y)* aufzurufen und dadurch den String *str* im Grafikfenster an der Position *(x, y)* auszugeben. Das Koordinatenpaar *(x, y)* bezeichnet dabei das linke Ende der *Basislinie* des ersten Zeichens in *str*.
Die Art und Größe der Schrift kann während der Programmierung in der Eigenschaft *font* festgelegt werden. Leider werden hier keine Escape-Sequenzen wie etwa "/n" für Zeilenumbruch erkannt.

Aufgabe: erstelle folgende Grafik:



Lösung

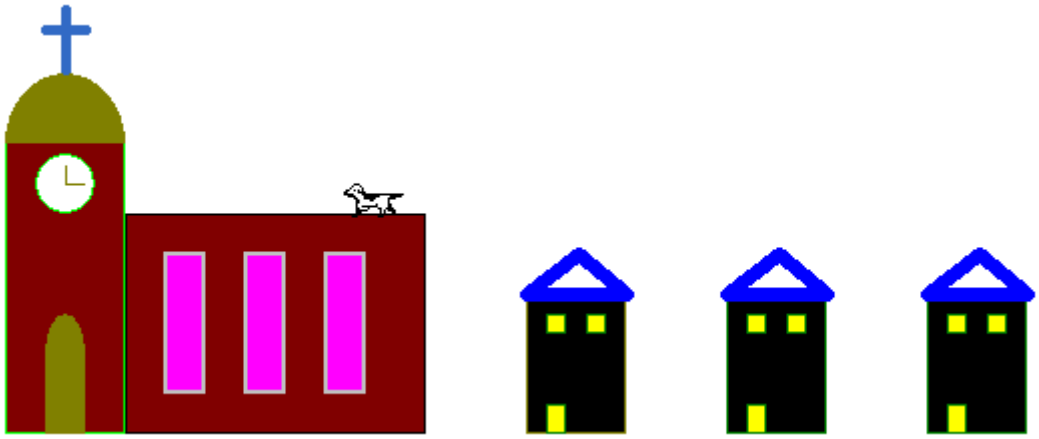
```
import java.awt.*;
…….
private void jbZeichneMouseClicked(java……) {

  Graphics zeichenagent =panZeichenbrett.getGraphics();
  zeichenagent.drawRect(10,20,30,60);

  zeichenagent.setColor(Color.red);
  zeichenagent.fillRect(50,20,30,60);

  zeichenagent.setColor(Color.black);
  zeichenagent.drawOval(90,20,60,60);

  zeichenagent.setColor(Color.yellow);
  zeichenagent.fillOval(160,20,30,60);

  int[] arx = {200,250,225};
  int[] ary = {20,20,80};
  zeichenagent.setColor(Color.BLUE);
  zeichenagent.drawPolygon(arx,ary,arx.length);

  int [] arx2 = {250,275,300};
  int [] ary2 = {80,20,80};
  zeichenagent.setColor(Color.BLUE);
  zeichenagent.fillPolygon(arx2,ary2,arx2.length);

  zeichenagent.setColor(Color.BLACK);
  int x = 10;
  for (int i=0; i<60; ++i) {
     zeichenagent.drawLine(x,90,x,150);
     x += 1+3*Math.random();
  }

  zeichenagent.setColor(Color.GREEN);
  zeichenagent.fillArc(150,90,60,60,0,90);
  //Hinweis: Umfang später zeichnen als Inhalt!
  zeichenagent.setColor(Color.BLACK);
  zeichenagent.drawArc(150,90,60,60,-150,270);

  zeichenagent.drawString("Fertig", 250, 120);
}
```

Aufgaben

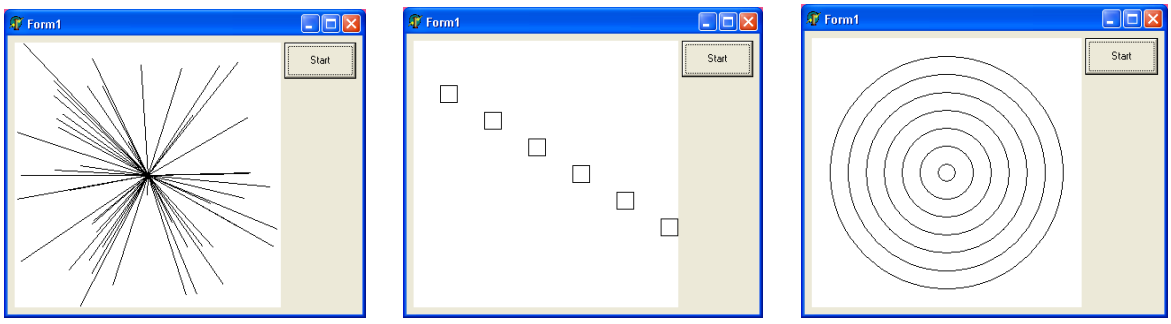
1. Erstelle farbige reguläre n-Ecke, deren Umrandung eine andere Farbe besitzt als die Fläche.

2. Erstelle ein DIN-A-4 großes Bild mit kariertem Muster (also ein Blatt eines Rechenheftes). Jedes Rechteck soll mit einer der 5 Farben rot, gelb, grün, blau, weiß gefüllt sein.

3. Zeichne folgendes Bild:



4. Zeichne folgende Bilder:



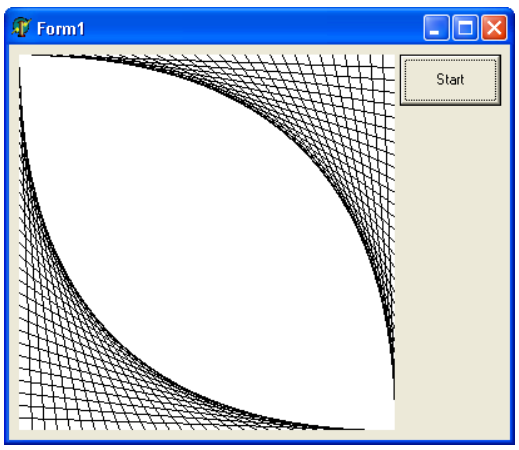
5. Erstelle ein Programm, welches auf der Zeichenfläche des Bildes ein Blatt eines Hausheftes darstellt. Arten des Hausheftes: kariert, liniert, Notenlinien. Das Blatt soll links und rechts einen Rand haben.

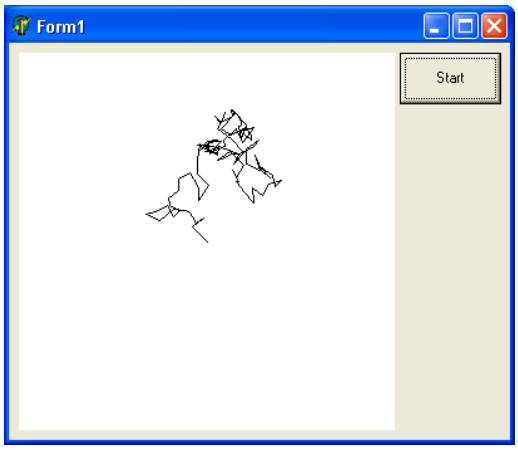
6. An zehn zufälligen Orten der Bildfläche soll jeweils ein kleines Häuschen gezeichnet werden.

7. a) Erzeuge das linke Bild! Sorge dafür, dass in deinem Bild die Lücken oben

links und unten rechts geschlossen sind.

b) Erzeuge den rechten Zufallsweg!





8. Zeichne 20 ineinander verschachtelte Rechtecke ! Hinweis: Es müssen zuerst die großen, danach die kleineren Rechtecke gezeichnet werden, weil ein Rechteck alles in seinem Inneren übermalt.

9. Zeichne 10 Kreise und Rechtecke, die alternierend ineinander verschachtelt sind und sich berühren! Hinweis: Zwei aufeinanderfolgende Kreisradien unterscheiden sich um den Faktor $\sqrt{2}$

10. Zeichne 15 zufällig große Kreise an zufälligen Orten!

Animation

Folgendes Programm lässt ein Rechteck über den Bildschirm (Panel) laufen:

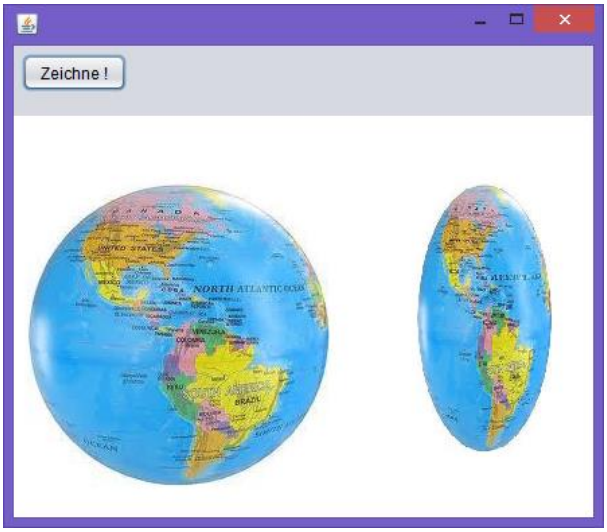
```
private void btZeichneMouseClicked(java…..) {
  panZeichenbrett.setBackground(Color.WHITE);
  Graphics zeichenagent = panZeichenbrett.getGraphics();
  int y = 20;
  for (int x = 10; x<378; x++) {
     zeichenagent.setColor(Color.BLACK);
     zeichenagent.drawRect(x,y,30,60);
     warte(50);
     zeichenagent.setColor(Color.WHITE);
     zeichenagent.drawRect(x,y,30,60);
  }
}
```

Aufgaben

1. Ein kleiner Kreis soll 20 mal an zufälligen Orten des Bildschirms erscheinen!

2. Ein kleiner Kreis soll sich 50 mal auf dem Bildschirm in eine zufällige Richtung hin bewegen.

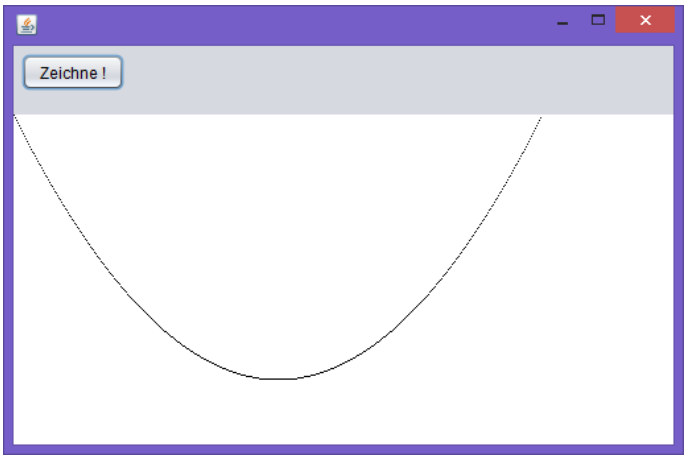
Grafik einfügen




```
private void btZeichneMouseClicked(java.....) {
  panZeichenbrett.setBackground(Color.WHITE);
  Graphics zeichenagent = panZeichenbrett.getGraphics();
  Image einBild = new ImageIcon("Globus.JPG").getImage();
  zeichenagent.drawImage(einBild,10,50,null);
  // An Stelle x=10 und y=50 einfügen
  zeichenagent.drawImage(einBild,300,50,100,200,null);
  // An Stelle x=300 und y=50 einfügen mit der Breite 100 und der Höhe 200
}
```

Funktionsgraph

Folgendes Programm zeichnet eine parabelförmige Kurve:




```
private void btZeichneMouseClicked(java……) {
  panZeichenbrett.setBackground(Color.WHITE);
  Graphics zeichenagent = panZeichenbrett.getGraphics();
  zeichenagent.setColor(Color.BLACK);
  int y;
  for (int x = 0; x<400; x++) {
    y = Math.round (-(x-200)*(x-200)/200)+200;
    zeichenagent.drawLine(x,y,x,y);
  }
}
```