

1. Zeichen, Zeichenketten und Texte in Anwendungen

Neben der Computergrafik ist der Umgang mit Texten die Hauptaufgabe von Computerprogrammen. Texte treten dabei in unterschiedlicher Form auf:

- Es kann sich um *einzelne Zeichen* handeln, die bestimmte Aktionen auslösen sollen oder z. B. zu Zahlen zusammengesetzt werden.
- Aufeinanderfolgende zusammengehörende Zeichen bilden *Zeichenketten*, sogenannte *Strings*, die z.B. Erläuterungen zum Zustand des Programms oder Daten (Nachname, Vorname, ...) enthalten.
- Mehrere Zeichenketten bilden *Texte* im üblichen umgangssprachlichen Sinn, die bearbeitet, gespeichert und wieder geladen werden.

Die *Bedeutung* solcher Zeichenkombinationen (von *Literalen*) ist erst einmal offen. Spezielle Zeichenfolgen bilden *Zahlen*, andere Postleitzahlen usw. Diese *Interpretation* der Zeichenketten muß besonderen Programmteilen überlassen werden, die meist entsprechende *Typumwandlungen* bewirken.

2. Die Eingabe einzelner Zeichen

Wenn eine Taste der Tastatur gedrückt wird, dann ist noch nicht klar, welches der laufenden Programme das Ergebnis dieses Tastendrucks auswerten soll. Allerdings hat immer nur ein Element des Windows-Bildschirms den *Fokus*, ist also ausgewählt. (Meist wird ein Element durch einen Mausklick oder durch das Drücken der Tabulatortaste vom Benutzer ausgewählt; manchmal steuert aber auch ein Programm den Fokus.) Dieses Element kann dann auf Ereignisse reagieren, die von der Tastatur ausgelöst werden. Standardereignisse sind:

- *keyDown* und *keyUp* mit denen der Zustand von Tastatur und Maus beim Drücken bzw. Loslassen einer Taste abgefragt werden kann. Auch Sondertasten und Tastenkombinationen werden angezeigt.
- *keyPress*, mit dessen Hilfe das eingegebene Zeichen ermittelbar ist.

Zeichen werden wie alle anderen Größen als Bitfolgen gespeichert. In Java werden zwei Byte (also 16 Bit) zur Zeichendarstellung verwandt. Da darin $2^{16} = 65536$ verschiedene Zustände möglich sind, können auch 65536 verschiedene *Unicode*-Zeichen benutzt werden – es sind also so ziemlich alle Zeichensätze und Sprachen darstellbar. In anderen Systemen wird oft nur eine 8-Bit-Zeichendarstellung benutzt, weil ältere Rechner dieses Format benutzen. Das achte Bit wird manchmal auch zur Kontrolle der Richtigkeit der Übermittlung benutzt. In beiden Fällen sind nur 128 unterschiedliche Zeichen möglich, und damit führen nationale Sonderzeichen wie die deutschen Umlaute zu Fehlern.

In Java schließt man Zeichen in Hochkommas ein (z. B. als `'a'`). Davon gibt es verschiedene. Nehmen Sie das über dem # auf der Tastatur.

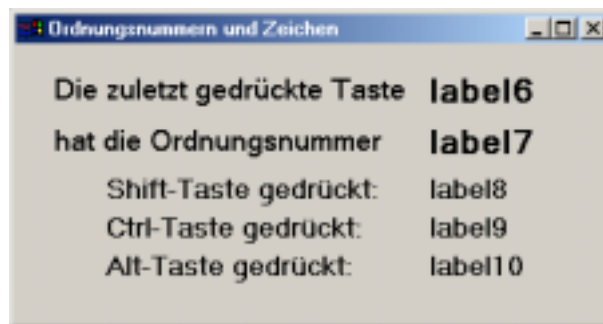
Welches Zeichen nun welcher Bitfolge entspricht, wird durch den benutzen *Zeichencode* festgelegt. Oft wird der American Standard Code for Information Interchange benutzt, den man als *ASCII* abkürzt. Die ersten 128 Zeichen dieses Codes enthalten große und kleine Buchstaben, Ziffern, Satzzeichen des amerikanischen Alphabets sowie einige Sonderzeichen für die Wagenrücklauf- (*Return*)-Taste (Zeichen 13), die *ESC*-Taste (Zeichen 27) und andere. Die Bezeichnung der Sonderzeichen stammt teilweise noch aus den Fernschreiber-Zeiten, in de-

nen z. B. das Zeichen *Bell* (Zeichen 8) die Fernschreibklingel läuten ließ. (Heute bewirkt es meist einen Piepton.) Windows benutzt einen *ISO-Code* (von *International Organization for Standardization*), der bei Zeichen mit Ordnungsnummern ab 128 vom ASCII abweicht und z. B. nationale Sonderzeichen enthält.

Die ASCII-Tabelle, in der die Zuordnung von Zeichen zu ihren Ordnungsnummern aufgeführt ist, findet man in fast allen Programmierhandbüchern. Viel einfacher ist es aber, sich die Nummern der gerade benötigten Zeichen von einem Programm ausgeben zu lassen. Man drückt die gewünschte Taste und liest die Ordnungsnummer ab.

Der 16-Bit-Datentyp *Zeichen* wird in Java als *char* bezeichnet. Die Ordnungsnummer eines Zeichens ermittelt man über eine Typumwandlung (type-casting): Hat man ein Zeichen *c*, dann liefert (int) *c* seine Ordnungsnummer. (Dabei können sehr große Ordnungsnummern auch negative Zahlen liefern.) (Umgekehrt liefert der Aufruf (char) *i* das Zeichen mit der Ordnungsnummer *i*.) Wollen wir die ermittelte Nummer (eine ganze Zahl) wieder am Bildschirm darstellen, dann müssen wir sie z. B. mit der Methode *valueOf* der *String*-Klasse in eine Zeichenkette verwandeln: `label1.setText(String.valueOf(i));`

Wir wollen die Zeicheneingabe vom Formular selbst auswerten lassen. Dazu fügen wir zehn Label-Komponenten ein, von denen fünf einen erläuternden Text enthalten (Setzen Sie entsprechend die Eigenschaft *text* der Komponenten.), die sechste das Zeichen selbst, die siebente die Ordnungsnummer des Zeichens und die anderen angeben, welche Sondertasten gedrückt wurden.



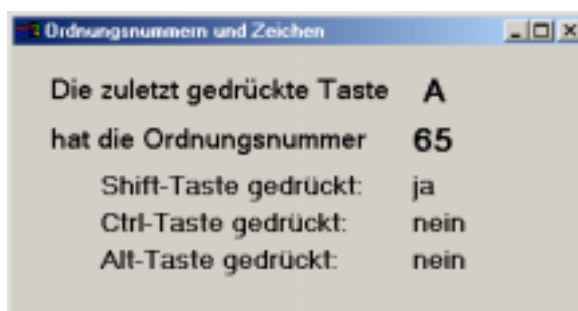
Die Ordnungsnummer soll ermittelt werden, wenn eine Taste losgelassen wird. Wir wählen deshalb das *keyDown*-Ereignis des Formulars und ordnen ihm eine Methode *Ordnungsnummer* zu.

```
private void Ordnungsnummer(Object source, KeyEvent e)
{
    label6.setText(" " + (char)(e.getKeyCode()));
    if (e.getShift()) label8.setText("ja"); else label8.setText("nein");
    if (e.getControl()) label9.setText("ja"); else label9.setText("nein");
    if (e.getAlt()) label10.setText("ja"); else label10.setText("nein");
    label7.setText(String.valueOf(e.getKeyCode()));
}
```

Der Text-Eigenschaft der Label-Komponente wird ein Wert zugewiesen (und so dargestellt), ...

der mithilfe der String-Klasse aus einer Zahl gewonnen wird, ...

... die der Codenummer der Zeichentaste entspricht.



Wir ermitteln damit einige Ordnungsnummern und den Zustand der Shift-Taste:

| Zeichen | Code | Shift |
|---------|------|-------|
| `` | 32 | false |
| `A` | 65 | true |
| `.` | 190 | false |
| `a` | 65 | false |

| Zeichen | Code | Shift |
|---------|------|-------|
| `0` | 48 | false |
| `?` | 219 | true |
| F1 | 48 | 112 |
| ← | 37 | false |

| Zeichen | Code | Shift |
|---------|------|-------|
| return | 13 | false |
| esc | 27 | false |
| → | 39 | false |
| ↑ | 38 | false |

3. Aufgaben

1. Erzeugen Sie ein Formular und darauf eine Label-Komponente namens *Ausgabe*. Ergänzen Sie das Programm so, dass
 - a: nur große Buchstaben im erscheinen.
 - b: nur Buchstaben, aber keine anderen Zeichen eingegeben werden können.
 - c: bei Eingabe eines beliebigen Zeichens ein Stern „*“ im Eingabefeld erscheint.
 - d: die eingegebenen Zeichen „rückwärts“ dargestellt werden, also das zuletzt eingegebene Zeichen ganz links.

2. Zahlen kann man aus einzelnen Zeichen zusammensetzen: Immer wenn eine neue Ziffer eingegeben wird, verschiebt man die alten um eine Stelle „nach links“ (multipliziert sie also mit 10) und hängt die neue Ziffer „rechts“ an. Den Wert einer Ziffer erhält man durch einen einfachen Trick: Da die Ziffern „0“ bis „9“ im ASCII aufeinander folgen, zieht man die Ordnungsnummer der eingegebenen Ziffer von der Ordnungsnummer der „0“ ab. Wird ein Zeichen eingegeben, das keine Ziffer ist, dann ist die *Zahleneingabe* beendet.

| | | |
|---|--|-----------------------------------|
| Zahl ← 0 | | |
| Lies ein Zeichen von der Tastatur | | |
| SOLANGE das letzte Zeichen eine Ziffer war TUE | | |
| <table border="1" style="width: 80%; margin-left: 20px;"> <tr> <td style="padding: 5px;">Zahl ← 10*Zahl + ORD(Zeichen) – Ord(`0`)</td> </tr> <tr> <td style="padding: 5px;">Lies ein Zeichen von der Tastatur</td> </tr> </table> | Zahl ← 10*Zahl + ORD(Zeichen) – Ord(`0`) | Lies ein Zeichen von der Tastatur |
| Zahl ← 10*Zahl + ORD(Zeichen) – Ord(`0`) | | |
| Lies ein Zeichen von der Tastatur | | |

- a: Realisieren Sie das Verfahren.
- b: Führen Sie eine Korrekturmöglichkeit ein: Die ← -Taste löscht die letzte eingegebene Ziffer!
- c: Ermöglichen Sie die Eingabe von ganzen Zahlen mit Vorzeichen.
- d: Ermöglichen Sie die Eingabe von Gleitpunktzahlen, den Zahlen mit einem Nachkommateil.

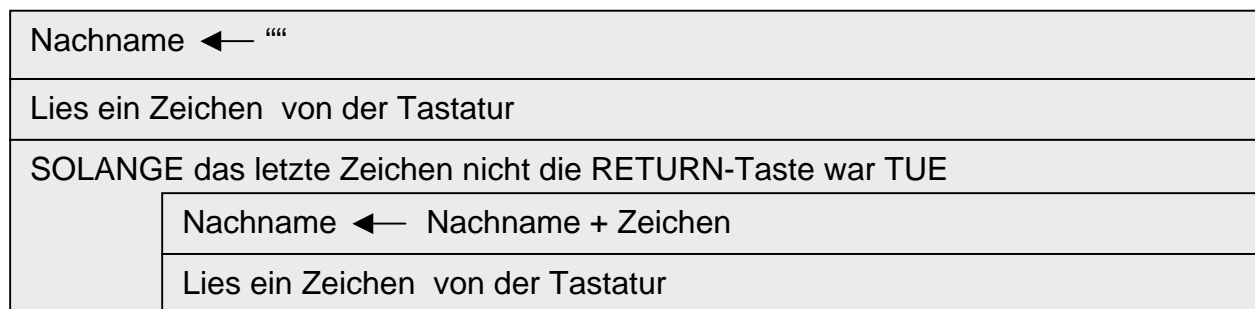
3. In vielen Fällen kann man Fehler schon bei der Eingabe vermeiden, indem nur **zulässige Werte** vom Programm angenommen werden. Führen Sie Editierfelder ein, die
 - a: nur Datumswerte der Form *tt.mm.jj* annehmen.
 - b: Dezimalzahlen automatisch in Prozentwerte umformen und anzeigen.
 - c: nur Schiffspositionen in der Form *nnn°mm′ss″* (Grad-Bogenminuten-Bogensekunden) annehmen.

4. Eingabe und Bearbeitung von Zeichenketten

Im Gegensatz zu einzelnen Zeichen will man bei der Eingabe von ganzen Zeichenketten nicht auf Korrekturmöglichkeiten verzichten. Erst nach der Eingabe eines dafür vorgesehenen Zeichens – der *RETURN-Taste* (Zeichen 13) – soll die Eingabe an das System „abgeschickt“ werden. Einige Sonderzeichen wie ← (Zeichen 8) und <Entf> (Zeichen 46) dienen zur Korrektur der vorherigen Eingabe. In der Zwischenzeit muss die „Eingabezeile“ aber irgendwo bleiben.

Zeichenketten werden in Java in Variablen des Typs *String* gespeichert. Stringlitterale werden durch Anführungszeichen eingeschlossen (nicht durch Hochkommas). Die aktuelle Länge des Strings wird von der Methode *length()* ermittelt. (Sie kann einige Tausend Zeichen betragen, obwohl das nicht sehr sinnvoll ist.) Das *i*-te Zeichen innerhalb des Strings kann mit der Methode *charAt(..)* erhalten werden. (Die Zählung beginnt bei 0.) Hat also eine String-Variable *nachname* den Wert *Meier*, dann erhält man das „i“ durch den Aufruf *nachname.charAt(2)*; Man kann auch Zeichen ändern: Der Befehl *s = nachname.replace('i', 'y')* macht aus Herrn *Meier* einen Herrn *Meyer*.

Eine einfache Möglichkeit zur Zeichenketteneingabe ist es also, den benötigten String einfach aus den eingegebenen Zeichen „zusammenzubasteln“. Wir wollen uns den Wert einer Variablen „Nachname“ zusammensuchen:



Hat man solch einen String erst einmal im Speicher, dann kann eine ganze Reihe von *Stringbearbeitungsfunktionen* auf ihn angewandt werden:

- Die Methode *indexOf(char)* liefert den Index des ersten Auftretens des Zeichens im String. Wird der Teilstring nicht gefunden, erhält man den Wert -1.
Beispiel: *s = "Meier"; s.indexOf('i');* liefert 2.
- Die Methode *substring(int,int)* liefert einen String mit den Zeichen zwischen den angegebenen Positionen. Zu beachten ist, dass die zweite Zahl den Index nach dem letzten gewünschten Zeichen angibt.
Beispiel: *s = "Meiers"; s.substring(1,5);* liefert "eier".

- Weil Strings in Java als *Referenztypen* implementiert werden, enthält eine Stringvariable nicht den Wert des Strings, sondern seine Adresse. Vergleicht man also z. B. zwei Strings, dann werden ihre Adressen verglichen, nicht ihre Inhalte! Zeichenketten vergleicht man mit der Methode `equals(..)`.

Beispiel: `s = "Meier"; s.equals("Meyer")` liefert `false`.

Etwas komfortabler gestaltet sich die Eingabe von Strings, wenn wir *Editierfelder* benutzen. Falls deren Eigenschaft `readOnly` den Wert `false` hat (Voreinstellung), können wir die dargestellte Textzeile bearbeiten, also auch während des Programmlaufs eingeben und im Programm auswerten. Der eingegebene Text findet sich in der `text`-Eigenschaft des Feldes.

Entwerfen wir also ein Formular mit zwei Editierfeldern und einem Button. In dessen Ereignisbehandlungsmethode wird er Inhalt des oberen Textfeldes in eine Stringvariable kopiert. Dann wird „irgendetwas“ (hier: Verwandlung in Großschrift) damit gemacht. Das Ergebnis wird im unteren Textfeld dargestellt.

```
private void button1_click(Object source, Event e)
{
    String s=edit1.getText();
    s = s.toUpperCase();
    edit2.setText(s);
}
```

... und ausgeben

Text aus dem Editierfeld fischen, ...

... umwandeln ...

The screenshot shows a window titled "Form1" with a standard Windows-style title bar. Inside the window, there are two text input fields. The top field is labeled "Eingabe:" and contains the text "das ist alles kleingeschrieben". Below this field is a button with the text "Umwandeln". The bottom field is labeled "Ausgabe:" and contains the text "DAS IST ALLES KLEINGESCHRIEBEN".

5. Ausgabe von Zeichenketten

Zeichenketten könne praktisch überall dort ausgegeben werden, wo eine „Beschriftung“ möglich oder erforderlich ist, also

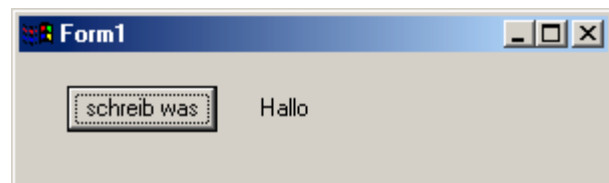
- als *text*-Eigenschaft von Komponenten, vor allem der *label*-Komponenten, die gerade zur Anzeige von Zeichenketten gedacht ist. Haben wir eine Label-Komponente namens *Anzeige* auf einem Formular *form1* und einen auszugebenden Text in einer Variablen *s*, dann zeigt der Befehl

```
form1.Anzeige.text.setText(s);
```

diesen Text an.

- als Ergebnis der *drawString*-Methode eines Grafik-Kontextes. Diese „zeichnet“ den Text ab der angegebenen Position. Wollen wir z. B. auf der *Graphics*-Komponenten *g* eines Formulars *form1* schreiben, dann funktioniert

```
g.drawString("Hallo",120,20);
```



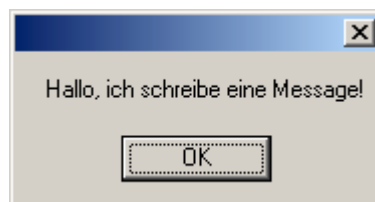
```
private void button1_click(Object source, Event e)
{
    Graphics g = createGraphics();
    g.drawString("Hallo",120,20);
}
```

- als Inhalt der *text*-Eigenschaft eines Editierfeldes, dessen *readOnly*-Eigenschaft bei Bedarf auf *TRUE* gesetzt wird (s.o.).

Unabhängig von den Komponenten eines Formulars können Texte auch in eigenen *Meldungsfenstern* angezeigt werden.

- Am einfachsten geschieht das mit der Methode *show* einer *MessageBox*, die eine Textzeile in der Mitte des Bildschirms anzeigt. Wir wollen so eine Meldung beim Mausklick auf einen Knopf namens *button1* erscheinen lassen:

```
private void button1_click(Object source, Event e)
{
    MessageBox.show("Hallo, ich schreibe eine Message!");
}
```



6. Texte, Stringlisten und Dialoge

Will man mehr als eine Textzeile anzeigen oder bearbeiten, dann benötigt man Komponenten, die entsprechenden Speicherplatz sowie die Methoden zum Umgang mit diesen Texten bereitstellen. Einige Objekte, z. B. die *RichEdit*-Komponente, verfügen über eine solche Eigenschaft, die man *Lines* nennt. Es handelt sich dabei um eine *Stringarray*, also ein Feld, das mehrere Strings aufnehmen kann. Diese können von den Komponenten aufgenommen, ihrer Anordnung verändert, in einer Datei gespeichert und wieder von dort geladen werden. Als Beispiel wollen wir eine RichEdit-Komponente etwas genauer ansehen.

Eine RichEdit-Komponente *enthält* einerseits die Strings, andererseits *manipuliert* sie diese, indem sie die Strings am Bildschirm darstellt sowie Eingaben und/oder Löschvorgänge des Benutzers auf die Strings überträgt. Sie stellt eine Art „Tor“ zu den „Zeilen“ dar.

Wollen wir weitergehende Eigenschaften wie das Speichern und Laden ausnutzen, dann müssen wir diese Aktionen selbst veranlassen. Dazu stehen die folgenden Methoden bereit:

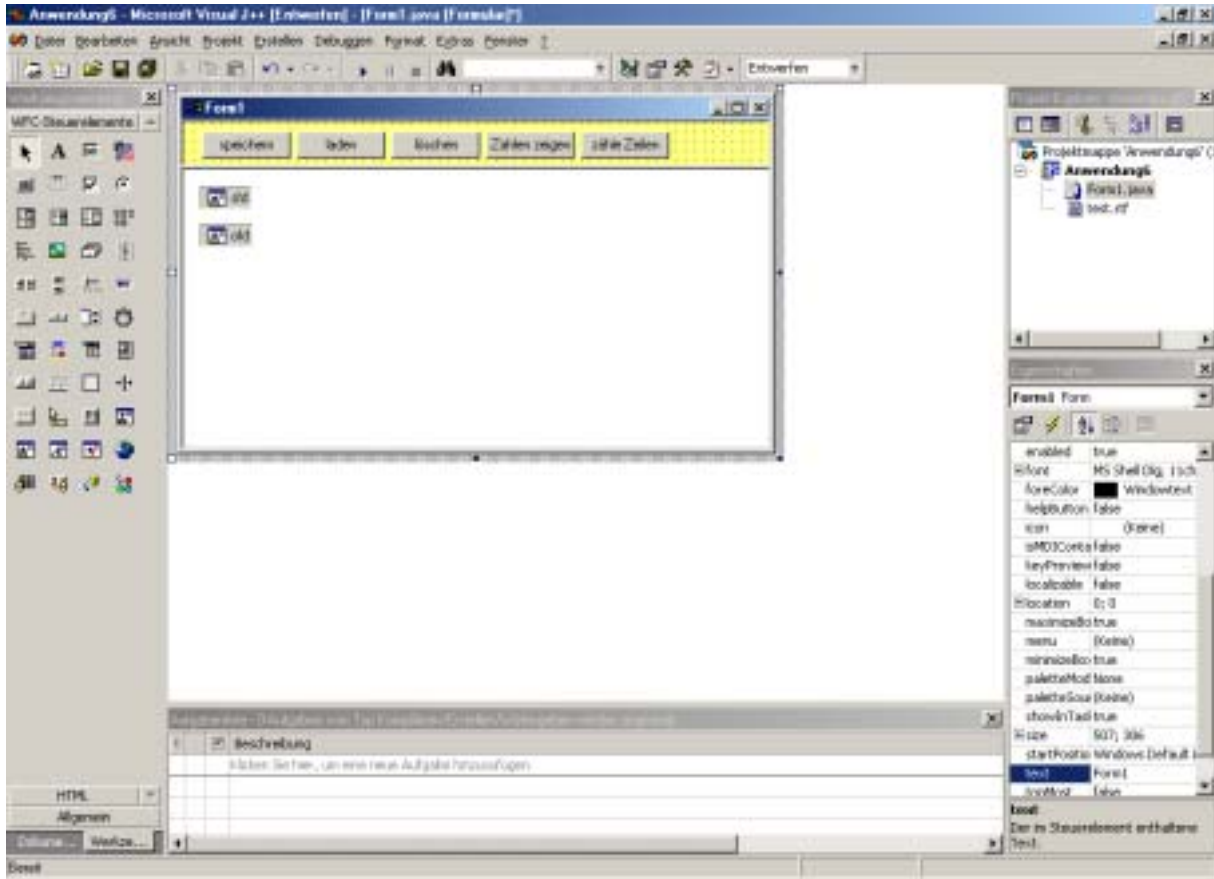
- Die Methode *clear* löscht den Inhalt der Stringliste.
- Die Methode *loadFile(filename)* lädt den Inhalt einer RTF-Datei in die Stringliste. (RTF bedeutet „rich text format“)
- Die Methode *saveFile(filename)* speichert die Stringliste in einer RTF-Datei.

Als kleines Anwendungsbeispiel wollen wir einen „Mini-Editor“ schreiben, der eine RichEdit-Komponente verwaltet. Als „Leckerbissen“ wollen wir den darin enthaltenen Text – wie in Windows üblich – laden und speichern, indem entsprechende *Dialogboxen* geöffnet werden, die eine Auswahl des Dateinamens ermöglichen. Entsprechende Komponenten enthält J++ in der Werkzeugliste. Dort befinden sich u. a. die *Save-* und *Open-Dialoge* für Dateinamen. Sie sind während der Entwurfszeit am Bildschirm als kleine Symbole sichtbar, während des Programmlaufes aber nicht: Es handelt sich um *unsichtbare Komponenten*! Beide verfügen über eine Methode *getFilename*, die den gewählten Dateinamen liefert, und eine Methode *showDialog*, die den Dialog startet und eine ganze Zahl zurückgibt, die ausgewertet werden kann. Ihre Benutzung führt deshalb zu typischen Programmzeilen, z. B.:

```
int result = openFileDialog1.ShowDialog();  
if (result == DialogResult.OK) {...}
```

Wir entwerfen unseren Editor am Bildschirm, indem wir geeignete Buttons auf *einer Panel-Komponente* am oberen Bildschirmrand platzieren und eine *RichEdit-Komponente* den Rest des Formulars ausfüllen lassen (beides mit Hilfe der Eigenschaft *dock!*). Irgendwo bringen wir dann noch die beiden Dialogkomponenten unter.

Die Buttons dienen zum Speichern, Laden, Löschen des Textes. Als Beispiele dienen die weiteren Buttons zum Erzeugen und Einfügen einiger Textzeilen und zum Ermitteln der Zeilenzahl im Textfeld (das wir auch so nennen, also *textfeld*).



Wir gehen der Reihe nach vor:

Speichern:

```
private void speichern_click(Object source, EventArgs e)
{
    sfd.SetFileName ("test.rtf");
    sfd.SetFilter("RTF-Dateien (*.rtf)|*.rtf");
    int result = sfd.ShowDialog();
    if (result == DialogResult.OK )
    {
        String fileName = sfd.GetFileName();
        textfeld.SaveFile(fileName);
    }
}
```

vorgegebener Dateiname

Filter für Datei-Suffixe (ausprobieren!)

Dialog starten und auswerten

Editorinhalt speichern



Laden:

```
private void laden_click(Object source, EventArgs e)
{
    ofd.SetFileName("test.rtf");
    ofd.SetFilter("RTF-Dateien (*.rtf)|*.rtf");
    int result = ofd.ShowDialog();
    if (result == DialogResult.OK)
    {
        String fileName = ofd.GetFileName();
        textfeld.LoadFile(fileName);
    }
}
```

vorgegebener Dateiname

Filter für Datei-Suffixe

Dialog starten und auswerten

Editorinhalt laden

Löschen:

```
private void loeschen_click(Object source, EventArgs e)
{
    textfeld.Clear();
}
```

Textzeilen erzeugen und einfügen:

Dazu erzeugen wir ein String-Array, in das wir Zeilen einfügen. Dieses Feld wird dann in den Editor kopiert.

```
private void zahlenZeigen_click(Object source, EventArgs e)
{
    String[] zeilen = new String[20];
    for(int i=0;i<10;i++) zeilen[i] = "Zeile "+i;
    textfeld.SetLines(zeilen);
}
```

Textfeld erzeugen

Zeilen einfügen

in den Editor einfügen

Zeilen zählen:

Wir kopieren den Editorinhalt in ein String-Array, dessen Länge wir bestimmen und ausgeben.

```
private void drin_click(Object source, EventArgs e)
{
    String[] l = textfeld.GetLines();
    int h = l.Length;
    MessageBox.Show("drin sind "+h+" Zeilen!");
}
```

Editorinhalt kopieren
und zählen