

## Dateizugriff unter Java

### Inhalt:

1. Einführung
2. Arbeit mit Textdateien
  - 2.1. Zeilenenden von Textdateien
  - 2.2. Schreiben in eine Textdatei
  - 2.3. Beispiel: Liste von Quadratzahlen in einer Textdatei
  - 2.4. Anhängen an eine Textdatei
  - 2.5. Einlesen einer Textdatei
  - 2.6. Beispiel: Änderung eines Datei-Inhalts
  - 2.7. Praxis-Beispiel: Erzeugung eines geeigneten Klartextes
3. Erweiterungen
  - 3.1. Datei-Operationen
  - 3.2. Zugriff auf Dateien mithilfe eines JTextAreas
  - 3.3. Auswahl eines Dateinamens während des Programmlaufs
  - 3.4. Ausblick: Random-Access-Dateien
4. Aufgaben

## Einführung

Der Zugriff auf Dateien ist ein Spezialfall für die Anwendung von sogenannten Streams (Datenströmen). Ein Stream ist ein Informationsfluss zwischen einer Quelle und einem Ziel, dessen Ende und damit dessen Länge nicht zu Beginn abgeschätzt werden kann. Als Quelle oder Ziel eines Streams können neben einer Datei auf der Festplatte auch das Internet, der Bildschirm, die Tastatur etc. infrage kommen.

Voraussetzung für den Zugriff auf Dateien ist, dass der Compiler die entsprechenden Java-Klassen (für Input-Output-Operationen) einbindet:

```
import java.io.*; // für Dateizugriffe
```

Diese Zeile muss zu Beginn des Programms stehen.

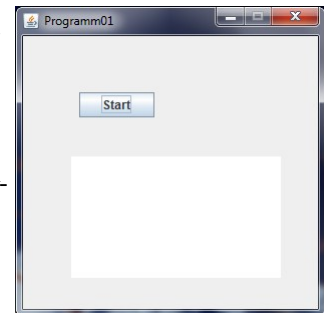
Da bei einem Zugriff auf eine Datei Fehler auftreten können (z. B. Datei nicht vorhanden, keine Lese- bzw. Schreibberechtigung, u.s.w.), wird ggf. bei einem entsprechenden Methodenaufruf eine `IOException` erzeugt, die durch das Programm abgefangen werden muss. Ein Java-Programmblock, der auf Dateien zugreift, sieht deshalb *immer* wie folgt aus:

```
try {  
    // Anweisungen für die Dateioperationen  
} catch (IOException e) {  
    // Hier können weitere Anweisungen für den Ausnahmefall stehen  
}
```

Im Folgenden stelle ich nur noch die Programmteile innerhalb des `try`-Blocks dar.

Da als Datei-Zugriff auch das *Schreiben* in eine Datei möglich sein soll, ist die Verwendung eines Applets nicht möglich, weil Applets (zumindest bei Ausführung in einem Browser<sup>1</sup>) keinen verändernden Zugriff auf das System haben (sollen).

Weitere Voraussetzung ist also, dass eine Java-Anwendung erstellt wird.<sup>2</sup> Im einfachsten Fall ist das ein `JFrame` mit einem `Button` und einer `TextArea` als Ausgabemöglichkeit (siehe Abbildung).



Wenn innerhalb eines Java-Quelltextes auf eine Datei mit einem absoluten Speicherort auf der Festplatte zugegriffen werden soll, muss der Backslash als „/“ oder als „\\“ geschrieben werden: „D:/Computer/test.txt“ oder „D:\\Computer\\test.txt“.

Wenn auf eine Datei im aktuellen Verzeichnis zugegriffen werden soll, muss diese

- ◆ bei der Ausführung des Programms vom Java-Editor aus im Verzeichnis der Java-Dateien,
- ◆ bei der Ausführung des Programms von der NetBeans-IDE aus im Projektverzeichnis bzw.
- ◆ bei der Ausführung der erzeugten JAR-Datei mit dieser im gemeinsamen Verzeichnis stehen.

1 Nicht irritieren lassen: Wenn ein Applet im Applet-Viewer ausgeführt wird, klappt der Zugriff noch hervorragend. Sobald die Ausführung desselben Applets im Browser stattfindet, ist dies jedoch nicht mehr möglich!

2 Details dazu im VLIN-Dokument „Java-Anwendungen und Zeichenketten“ von Eckart Modrow

## Arbeit mit Textdateien

Eine Textdatei enthält einen Text ohne weitere Formatierungen (wie z. B. Schriftarten, Schriftschnitte oder Schriftgrößen). Bis auf die Kennzeichnung des Zeilenendes (siehe unten) sowie eventueller Tabulator-Zeichen enthält eine Textdatei also nur Zeichen mit den ASCII-Codes 32 bis 254. In Windows kann man eine Textdatei am einfachsten mithilfe des Editors erstellen.

Textdateien haben üblicherweise die Dateiendung „txt“, aber hinter vielen anderen Dateiendungen verbergen sich auch nichts anderes als Textdateien: „asc“, „ini“, „htm“, „html“, „log“ und „sys“ sind ebenso typische Vertreter wie auch „java“, „pas“ etc. für Quelltexte von Programmen.

Im Informatik-Unterricht finden Textdateien vor allem in folgenden Bereichen Verwendung:

- ◆ Speichermöglichkeit z. B. für umfangreiche Berechnungsergebnisse
- ◆ Quellen für String-Manipulationen einschließlich „Datenbank“-Anwendungen und Kryptographie

## Zeilenenden von Textdateien

Textdateien bestehen im Wesentlichen aus Zeichen mit den ASCII-Codes 32 bis 254. Für das Ende einer Zeile muss also ein Symbol gewählt werden, das nicht in diesem Code-Bereich enthalten ist. In Anlehnung an mechanische Schreibmaschinen bzw. Fernschreiber, die ja auch Textdateien ausgeben sollten, wurden die beiden Zeichen „Carriage Return“ (Wagenrücklauf) und „Line Feed“ (Zeilen- bzw. Papier-Vorschub) gewählt. Dies entspricht den ASCII-Zeichen 13 und 10. Nachstehende Tabelle gibt eine Übersicht:

	ASCII	Java	
CR (CarriageReturn)	13	'\r' <sup>1</sup>	( <b>char</b> ) 13
LF (LineFeed)	10	'\n'	( <b>char</b> ) 10

In Windows-Textdateien folgen also die ASCII-Zeichen 13 und 10 (in dieser Reihenfolge) aufeinander, um das Ende einer Zeile zu kennzeichnen.

Linux-Textdateien verwenden als Kennzeichnung des Zeilenendes nur das ASCII-Zeichen 10, Mac-Textdateien nur das ASCII-Zeichen 13. Der Windows-Editor zeigt bei solchen von anderen Betriebssystemen erstellten Dateien anstelle eines Zeilenumbruchs ein kleines Rechteck an oder ignoriert diese. „Vernünftige“ Textverarbeitungsprogramme, zu denen ich ausnahmsweise auch einmal MS-Word zählen möchte, lesen diese Dateien jedoch problemlos ein.

## Schreiben in eine Textdatei

Für das (stückweise) Schreiben in eine Textdatei ist ein Objekt von Typ „FileWriter“ notwendig:

```
FileWriter fw = new FileWriter("Test.txt"); // FileWriter definieren
fw.write("Beispieltext"); // Zeichenkette in Datei schreiben
fw.close(); // Datei schließen
```

Zuerst wird dem FileWriter eine Datei zugeordnet, anschließend wird mithilfe dieses Writers ein Text in diese Datei geschrieben. Mithilfe der `close`-Methode wird der Schreibvorgang abgeschlossen und die Datei geschlossen.

1 Die Zeichenfolge „\r“ entspricht nur einem einzelnen Zeichen. Aus diesem Grund ist die Verwendung von einzelnen Anführungszeichen, die den Typ `char` kennzeichnen, gerechtfertigt. Wie bei anderen Zeichen vom Typ `char` dürfen diese allerdings auch in Strings, also in doppelten Anführungszeichen stehen.

Selbstverständlich dürfen mehrere `write`-Anweisungen hintereinander stehen.

Wenn dabei eine Zeile in der Textdatei beendet werden soll, gibt es folgende Möglichkeiten:

- ◆ An den letzten zu schreibenden String wird die Zeichenkette „`\r\n`“ angehängt:  
`fw.write("Zeilenende\r\n");`
- ◆ Die Zeichenkette „`\r\n`“ wird als separater String in die Datei geschrieben<sup>1</sup>:  
`fw.write("\r\n");`

Bei Linux bzw. Mac müssten entsprechend „`\n`“ bzw. „`\r`“ verwendet werden.

Wenn man anstelle des `FileWriters` einen `BufferedWriter` verwendet (Informationen dazu bitte dem WWW entnehmen), kann man den Befehl `bw.newLine();` verwenden.

Wenn in mehrere Dateien nacheinander geschrieben werden soll, kann der eingeführte `FileWriter` nach der Ausführung der `close`-Anweisung durch z. B. folgende Anweisung für die neue Datei wiederverwendet werden: `fw = new FileWriter("Test2.txt");`.

Wenn in mehrere Dateien gleichzeitig geschrieben werden soll, muss man mehrere `FileWriter`-Objekte verwenden.

### Beispiel: Liste von Quadratzahlen in einer Textdatei

In einer Textdatei soll die Liste der ersten zehn Quadratzahlen ausgegeben werden. Der Quelltext im `try`-Block sieht dann z. B. folgendermaßen aus:

```
FileWriter fw = new FileWriter("Ergebnisse.txt"); // Dateiname festl.
fw.write("Liste der Quadratzahlen:\r\n"); // Überschrift in Datei
for (int i=1; i<=10; i++) {
    fw.write("" + i + "*" + i + " = " + i*i + "\r\n");
} // der Reihe nach Zahlen und Quadrate in Datei schreiben
fw.close();
```

### Anhängen an eine Textdatei

Die oben dargestellte Variante des `FileWriters` sorgt dafür, dass grundsätzlich eine neue Datei erstellt wird – unabhängig davon, ob schon eine Datei mit diesem Namen existiert oder nicht. Ggf. wird die vorhandene Datei gelöscht. Durch eine kleine Modifikation können wir den `FileWriter` dazu bringen, an eine existierende Datei weitere Zeilen anzuhängen:

```
FileWriter fw = new FileWriter("Test.txt", true);
```

Ein `boolean`-Wert als zweiter Parameter im Konstruktor des `FileWriters` gibt an, ob Daten angehängt werden sollen oder nicht.

Wenn man das Anhängen davon abhängig machen will, ob schon eine Datei mit diesem Dateinamen existiert, bietet es sich an, den zweimal zu verwendenden Dateinamen in einer Variablen zu speichern und dann folgendermaßen vorzugehen:

```
String dateiname = "Test.txt";
FileWriter fw =
    new FileWriter(dateiname, (new File(dateiname)).exists());
```

### Einlesen einer Textdatei

Das Einlesen einer Textdatei ist wesentlich aufwendiger:

- ◆ Allein das Problem der unterschiedlich langen Dateien macht deutlich, dass nicht von vornherein feststehen kann, wie viele Zeichen gelesen werden können. Aus diesem Grund wird immer nur *ein* Zeichen gelesen.

<sup>1</sup> Die „sauberste“ Lösung, weil auf jedem System angeblich korrekt ausgeführt, soll die folgende sein: `fw.write(Character.LINE_SEPARATOR);`. Bei meinen Windows-Systemen funktionierte es allerdings nicht. (Lag wohl an Windows. ☹)

- ◆ Die eingelesenen „Zeichen“ werden als ASCII-Zahlenwerte zur Verfügung gestellt. Das Dateieinde entspricht der Zahl -1.

Das Einlesen muss also Zeichen für Zeichen erfolgen, wobei das eingelesene „Zeichen“ zuerst als Zahl abgespeichert und anschließend in ein Zeichen umgewandelt werden muss. Außerdem muss nach jedem gelesenen Zeichen überprüft werden, ob das Dateieinde erreicht ist.

Daraus ergibt sich folgendes Programmgerüst zum Einlesen einer Textdatei:

```
FileReader fr = new FileReader("Test.txt"); // Datei zum Lesen öffnen
char c;
int i = fr.read(); // erstes Zeichen als Zahl einlesen
while (i != -1) // solange das Dateieinde noch nicht erreicht ist, ...
{
    c = (char) i; // Zahl in das zugehörige Zeichen umwandeln
    // weiterer Programmtext, in dem "c" verwendet wird
    i = fr.read(); // nächstes Zeichen als Zahl einlesen
}
fr.close(); // Datei schließen
```

Häufig ist es wünschenswert, wenn eine ganze Zeile gelesen und das Ergebnis als String zur Verfügung gestellt wird. Dafür gibt es die Objekt-Klasse `BufferedReader`, deren Verwendung im Folgenden zu sehen ist:

```
BufferedReader br = new BufferedReader(new FileReader("Test.txt"));
String zeile = br.readLine(); // ganze Zeile auslesen
while (zeile != null) // solange das Dateieinde noch nicht erreicht ist
{
    textArea1.append(zeile+"\n"); // Zeile wird verwendet
    zeile = br.readLine(); // nächste Zeile wird eingelesen
}
br.close();
```

Das Ende der Datei wird also dadurch angezeigt, dass die eingelesene Zeichenkette den Wert „null“ hat. Bei einem `TextArea` genügt „\n“, um ein Zeilenende zu bewirken.

Das gleichzeitige Lesen aus einer Datei und schreiben in dieselbe Datei – also eine Modifikation der Daten einer Datei – ist nicht möglich. Entweder muss der Inhalt der eingelesenen Datei vollständig z. B. in einem `TextArea` zwischengespeichert werden (siehe unten), bevor die Schreiboperation beginnen kann, oder es muss zuerst eine temporäre Datei erstellt werden, die dann anschließend umbenannt wird (siehe unten).

### Beispiel: Änderung eines Datei-Inhalts

In einer Datei sollen alle Vokale verdoppelt werden. Aus „Berlin“ würde also „Beerliin“ werden. Umlaute lassen wir unverändert.

Als Grundlage benötigen wir zuerst eine Textdatei, die einen beliebigen Text enthält. Dazu öffnen wir den Windows-Editor (Programme | Zubehör | Editor) und kopieren dort einen beliebigen Text hinein, der z. B. dem WWW entnommen sein könnte. Diesen Text speichern wir als Datei „Vokale.txt“ beim Java-Editor im Verzeichnis der Java-Dateien und bei der NetBeans-IDE im Projektverzeichnis eines neuen Java-Projekts ab.

Da wir parallel Zeichen einlesen und schreiben müssen, werden sowohl `FileReader` als auch `FileWriter` benötigt:

```
FileReader fr = new FileReader("Vokale.txt");
FileWriter fw = new FileWriter("Vokale_neu.txt");
```

Die Ausgabe erfolgt also in eine Datei mit dem Namen „Vokale\_neu.txt“. Es handelt sich dann nicht um die selbe Datei; dazu wäre ein nachträgliches Löschen der Originaldatei und Umbenennung der neuen Datei notwendig (siehe unten).

Während des Ablaufs benötigen wir eine Integer-Variable zum Auslesen der „Zeichen“ sowie eine `char`-Variable für die Umwandlung dieser Zahl in ein Zeichen. Darüber hinaus wird die Ausgabe Stück für Stück in einer String-Variablen „zusammengebastelt“, die anfangs mit einem Leerstring initialisiert wird:

```
char c;
String ausgabe = "";
int i = fr.read();
```

Das erste Zeichen wird gleich mit ausgelesen.

Die folgende Schleife läuft, bis das Datei-Ende erreicht ist, und prüft zunächst, ob ein Zeilenende erreicht wurde. Wenn ja, wird der bisher zusammengebastelte Ausgabe-String mit einem Zeilenende in die Datei geschrieben und dessen Inhalt gelöscht:

```
while (i != -1) { // solange das Dateiende noch nicht erreicht ist
    if ( (i==10) || (i==13) ) { // Zeilenende-Zeichen?
        if (i==10) { // Hier nur Berücksichtigung von Windows-Dateien
            fw.write(ausgabe + "\r\n");
            ausgabe = "";
        }
    }
}
```

Anderenfalls handelt es sich bei dem eingelesenen Zeichen um ein „richtiges“ Zeichen, das jetzt daraufhin überprüft wird, ob es als Vokal doppelt angehängt werden soll:

```
else {
    c = (char) i;
    ausgabe = ausgabe + c; // Zeichen einmal anhängen
    if ( (c=='a') || (c=='e') || (c=='i') || (c=='o') || (c=='u') ||
        (c=='A') || (c=='E') || (c=='I') || (c=='O') || (c=='U') )
        ausgabe = ausgabe + c; // Zeichen ggf. ein zweites Mal anhängen
}
```

Abschließend wird das nächste Zeichen gelesen. Wenn das Datei-Ende erreicht ist, werden beide Dateien geschlossen:

```
i = fr.read();
} // Ende von while für Dateiende
fr.close();
fw.close();
```

Wenn man bei der Abfrage nach vielen Buchstaben die Aneinanderreihung mit „||“ vermeiden möchte, bietet sich folgende Anweisung an, die prüft, ob das Zeichen `c` in der angegebenen Zeichenkette aus den zu prüfenden Einzelbuchstaben enthalten ist:

```
if (("aeiouAEIOU").indexOf(c) > -1) ausgabe = ausgabe + c;
```

### Praxis-Beispiel: Erzeugung eines geeigneten Klartextes

Klartexte spielen in der Kryptographie eine entscheidende Rolle: Ein Klartext ist eine Botschaft, die in verschlüsselter Form an den Empfänger übermittelt werden soll. Damit die im Unterricht behandelten Verschlüsselungsverfahren (einschließlich der Cäsar-Verschlüsselung) problemlos anwendbar sind, können an den Klartext z. B. folgende Anforderungen gestellt werden:

- ◆ Ein Klartext darf ausschließlich Buchstaben enthalten. (Satzzeichen und Leerzeichen könnten einen Rückschluss auf den Inhalt zulassen.)

- ◆ Der Einfachheit halber sind nur Großbuchstaben zugelassen.
- ◆ Umlaute und das ß sind durch „AE“ etc. auszuschreiben.
- ◆ Jede Zeile bis auf die letzte soll genau 80 Zeichen enthalten, damit die Darstellung in einem Editor (ohne expliziten Zeilenumbruch) einfach möglich ist.

Wünschenswert wäre außerdem noch, dass eine automatische Umwandlung von Zahlen in Zahlworte, also z. B. Umwandlung von „574“ in „FUENFHUNDERTVIERUNDSEBZIG“ erfolgen würde.

Der Klartext der obigen Aufzählung würde also folgendermaßen aussehen:

```
EINKLARTEXTDARFAUSSCHLIESSLICHBUCHSTABENENTHALTENSATZZEICHENUNDLERZEICHENKOENNT
ENEINENRUECKSCHLUSSAUFDENINHALTZULASSENDEREINFACHHEITHALBERSINDNURGROSSBUCHSTABE
NZUGELASSENUMLAUTEUNDDASSSINDDURCHAEETCAUSZUSCHREIBENJEDEZEILEBISAUFDIELETZTESO
LLGENAUACHTZIGZEICHENENTHALTENDAMITDIEDARSTELLUNGINEINEMEDITOREINFACHMOEGLICHT
```

Zur Umwandlung benötigen wir als Grundlage wieder eine Textdatei, die einen beliebigen Text enthält und z. B. dem WWW entnommen sein könnte. Diesen Text speichern wir als Datei „Klartext.txt“ ab.

Der Ablauf ist jetzt im Prinzip so wie beim vorigen Beispiel beschrieben. Als Ausgabedatei wird jedoch „Klartext\_fertig.txt“ verwendet.

In der Schleife muss allerdings eine andere Art der Auswertung erfolgen. Es wird versucht, dem eingelesenen Zeichen ein Zeichen oder einen Ausdruck in Großbuchstaben von „A“ bis „Z“ zuzuordnen. Damit auch Sonderzeichen aus gängigen Fremdsprachen mit berücksichtigt werden können, ist die Auflistung etwas länger<sup>1</sup>:

```
while (i != -1)
{
    c = (char) i;
    if ((c >= 'A') && (c <= 'Z')) ausgabe = ausgabe + c;
    if ((c >= 'a') && (c <= 'z')) ausgabe = ausgabe + (char) (i-32);
    if (("äÄæE").indexOf(c) > -1) ausgabe = ausgabe + "AE";
    if (("öÖøEøØ").indexOf(c) > -1) ausgabe = ausgabe + "OE";
    if ((c == 'ü') || (c == 'Ü')) ausgabe = ausgabe + "UE";
    if (c == 'ß') ausgabe = ausgabe + "SS";
    if ((c >= 'À') && (c <= 'Ä')) ausgabe = ausgabe + 'A';
    if ((c >= 'à') && (c <= 'ä')) ausgabe = ausgabe + 'A';
    if ((c == 'å') || (c == 'Å')) ausgabe = ausgabe + 'A';
    if ((c >= 'È') && (c <= 'Ë')) ausgabe = ausgabe + 'E';
    if ((c >= 'è') && (c <= 'ë')) ausgabe = ausgabe + 'E';
    if ((c >= 'Ì') && (c <= 'Î')) ausgabe = ausgabe + 'I';
    if ((c >= 'ì') && (c <= 'î')) ausgabe = ausgabe + 'I';
    if ((c >= 'Ò') && (c <= 'Ï')) ausgabe = ausgabe + 'O';
    if ((c >= 'ò') && (c <= 'ï')) ausgabe = ausgabe + 'O';
    if ((c >= 'Û') && (c <= 'Û')) ausgabe = ausgabe + 'U';
    if ((c >= 'ù') && (c <= 'û')) ausgabe = ausgabe + 'U';
    if ((c == 'ç') || (c == 'Ç')) ausgabe = ausgabe + 'C';
    if ((c == 'ð') || (c == 'Ð')) ausgabe = ausgabe + 'D';
    if ((c == 'ñ') || (c == 'Ñ')) ausgabe = ausgabe + 'N';
    if ((c == 'š') || (c == 'Š')) ausgabe = ausgabe + 'S';
    if (("ýÝÿŸ").indexOf(c) > -1) ausgabe = ausgabe + 'Y';
    if (c == '%') ausgabe = ausgabe + "PROZENT";
    if (c == '‰') ausgabe = ausgabe + "PROMILLE";
    if (c == '€') ausgabe = ausgabe + "EURO";
    if (c == '$') ausgabe = ausgabe + "DOLLAR";
    if (c == '£') ausgabe = ausgabe + "PFUND";
    if (c == '¥') ausgabe = ausgabe + "YEN";
```

<sup>1</sup> An dieser Stelle sie ein kleiner Seitenhieb auf Java erlaubt: Mithilfe einer **switch**-Anweisung wäre das Ganze – wie z. B. unter Delphi möglich – im Prinzip wesentlich übersichtlicher zu gestalten gewesen. Java lässt jedoch bei **switch**-Anweisungen keine Abfrage von Bereichen zu, so dass diese etwas unglückliche **if**-Auflistung verwendet werden musste.

```

if (c == '&') ausgabe = ausgabe + "UND";
if (c == '@') ausgabe = ausgabe + "AT";
if (c == '$') ausgabe = ausgabe + "PARAGRAPH";
if (c == '©') ausgabe = ausgabe + "COPYRIGHT";
if (c == 'µ') ausgabe = ausgabe + "MUE";
if (c == '°') ausgabe = ausgabe + "GRAD";

```

Deutlich zu erkennen ist die Unterscheidung zwischen einzelnen Zeichen vom Typ **char** in einfachen Anführungszeichen und Zeichenketten vom Typ **String** in doppelten Anführungszeichen.

Im Wesentlichen wird bei der Umsetzung unterschieden zwischen Buchstaben-Bereichen, die mithilfe einer gemeinsamen Anweisung umgewandelt werden können, wie z. B. die Buchstaben zwischen „A“ und „Z“ oder zwischen „Ë“ und „Û“, und einzelnen Buchstaben – häufig mit Groß- und Kleinschreibung –, bei denen die jeweiligen Bedingungen durch „oder“ verknüpft werden oder mit „**indexOf**“ ausgewertet werden.

Noch nicht berücksichtigt wurden Zahlen, die wie die Nicht-Buchstaben-Zeichen einfach ignoriert werden. Zur Umwandlung dieser Zahlen in Zahlworte siehe Aufgabe 11.

Wenn der Ausgabestring die Länge von 80 Zeichen erreicht oder überschritten hat, können die ersten 80 Zeichen zusammen mit einem Zeilenende in die Ausgabedatei geschrieben werden. Der Ausgabestring ist dann entsprechend zu kürzen:

```

if (ausgabe.length() >= 80) {
    fw.write(ausgabe.substring(0,80) + "\r\n");
    ausgabe = ausgabe.substring(80);
}

```

Zu beachten ist bei der Methode „**substring**“ beim Aufruf mit zwei Parametern, dass der Substring von der erstgenannten Stelle – hier also von der Stelle „0“ – bis zur Stelle *vor* der zweitgenannten Stelle – hier also bis zur Stelle „79“ einschließlich – erzeugt wird.

Die restlichen Zeilen beenden die **while**-Schleife, schließen die zum Lesen geöffnete Datei, schreiben ggf. den noch vorhandenen Rest des Ausgabestrings in die Ausgabedatei und schließen auch diese:

```

    i = fr.read();
}
fr.close();
if (ausgabe.length() > 0) {fw.write(ausgabe + "\r\n");}
fw.close();

```

## Erweiterungen

Die folgenden Darstellungen sind nicht für die Arbeit mit Dateien notwendig, können das Leben aber an der einen oder anderen Stelle erheblich erleichtern.

### Datei-Operationen

Manchmal möchte man mit Java Dateien löschen oder umbenennen. Dafür bietet die File-Klasse die Methoden **delete** und **renameTo** an:

```

boolean erfolg = (new File(dateiname)).delete();
boolean erfolg = (new File(dateiname)).renameTo(new File("neu.txt"));

```

Beide Methoden liefern als Rückgabewert einen boolean-Wert, der anzeigt, ob die Operation erfolgreich war oder nicht.

In allen Fällen ist es selbstverständlich möglich, Variablen vom Typ „File“ zu verwenden:

```
File datei1 = new File("test.txt");
File datei2 = new File("neu.txt");
boolean erfolg = datei1.renameTo(datei2);
```

Dies hat neben der verbesserten Übersichtlichkeit den Vorteil, dass auch beim Java-Editor die Code-Vervollständigung angewendet werden kann.

Weitere interessante Methoden der File-Klasse sind folgende: `getAbsolutePath` (siehe unten), `isFile`, `isDirectory`, `length` und `mkdir`. Auf diese wird hier aber nicht weiter eingegangen.

## Zugriff auf Dateien mithilfe eines JTextAreas

Häufig möchte man den Inhalt einer Textdatei z. B. in einer TextArea angezeigt bekommen. Die zugehörigen Methoden existieren nur für die Swing-Variante (JTextArea), nicht jedoch für die AWT-Variante (TextArea). Beim „Zusammenklicken“ der Oberfläche muss also die entsprechende Auswahl getroffen werden.

Das Einlesen einer gesamten Textdatei in eine solche JTextArea erfolgt z. B. über folgende Anweisung, die wieder innerhalb eines `try`-Blocks stehen muss:

```
jTextArea1.read(new FileReader("Klartext.txt"), null);
```

Der Speicherort der Textdatei muss wie in der Einführung angegeben gewählt werden.

Die Schriftart innerhalb der JTextArea kann ggf. über die Eigenschaft „font“ eingestellt werden.

Analog zum Lesen erfolgt das Schreiben in eine Textdatei, wobei jetzt die Methode „write“ anstelle von „read“ und ein `FileWriter` anstelle des `FileReaders` verwendet wird:

```
jTextArea1.write(new FileWriter("Klartext_fertig.txt"));
```

Das Anhängen an eine vorhandene Textdatei ist mithilfe der oben beschriebenen Modifikation des `FileWriters` möglich.

Anstelle des neu definierten `FileReaders` bzw. `FileWriters` in den Argumenten von „read“ und „write“ könnten auch bereits vorher definierte und einer Datei zugewiesene Objekte „fr“ und „fw“ eingesetzt werden.

Bei der verwendeten JTextArea muss es sich nicht um ein visuelles Objekt im Programmfenster handeln. Es kann auch innerhalb einer Methode als temporäres Objekt erzeugt und verwendet werden:

```
JTextArea jTextArea2 = new JTextArea();
```

Beim Einlesen und Schreiben einer Textdatei muss bei der Verwendung einer JTextArea also kein Aufwand bezüglich der Zeilenenden getrieben werden; diese entsprechen den Zeilenenden der JTextArea und umgekehrt.

Der Zugriff auf die Daten innerhalb der JTextArea ist jedoch wesentlich komplizierter. Deswegen einige Hinweise dazu:

- ◆ Das Anfügen weiterer Zeilen am unteren Ende der JTextArea geschieht vergleichsweise einfach mithilfe der Methode „append“: `jTextArea1.append("weitere Zeile");`.
- ◆ Die Anzahl der Zeilen in der JTextArea erhält man ebenfalls sehr einfach mithilfe der Methode „getLineCount“.

- ◆ Der Zugriff auf bestimmte Zeilen innerhalb der JTextArea ist dafür wesentlich komplizierter:
  - Der Zugriff selbst erfolgt über die Methode „`getText(Start, Länge)`“. Dabei gibt „Start“ die Position des Zeichens innerhalb des gesamten Textes an. Da die Startposition außerhalb des vorhandenen Textes liegen könnte, muss diese Operation in einen `try-catch`-Block eingeschachtelt werden. Die zugehörige Exception „`BadLocationException`“ muss aber noch mit „`import javax.swing.text.*;`“ eingebunden werden.
  - Weiteres Problem ist das Finden der richtigen Startposition. Dafür gibt es die Methode „`getLineStartOffset(Zeilenummer)`“. Entsprechend gibt die zugehörige Methode „`getLineEndOffset(Zeilenummer)`“ die Position des Beginns der Folgezeile an, wobei Zeilenende-Zeichen mitgezählt werden.

Da die Zeilen mit 0 beginnend durchnummeriert werden, lautet der gesamte Programmtext für das Auslesen der dritten Zeile (also mit dem Index „2“) aus einer JTextArea und das Einfügen in ein JTextField folgendermaßen:

```
import javax.swing.text.*;
...
try {
    int start = jTextArea1.getLineStartOffset(2);
    int ende = jTextArea1.getLineEndOffset(2);
    jTextField1.setText(jTextArea1.getText(start, ende-start-1));
} catch (BadLocationException e2) {}
```

Die Länge des auszulesenden Bereichs ergibt sich dadurch, dass Zeilenende-Zeichen (in diesem Fall gehen wir von einem Zeilen-Ende-Zeichen aus) wegfallen müssen. Der Wert „`ende-start`“ muss also noch um 1 reduziert werden.

Wenn *alle* Zeilen der JTextArea der Reihe nach ausgelesen werden sollen, muss zusätzlich eine Zähl-Schleife verwendet werden, die auf „`getLineCount()`“ zurückgreift.

## Auswahl eines Dateinamens während des Programmlaufs

Die bisherige Darstellung geht davon aus, dass der Dateiname schon zum Zeitpunkt des Verfassens des Quelltextes bekannt ist. Das ist in der Regel jedoch nicht der Fall bzw. ein geschriebenes Programm sollte für mehr als eine Datei eingesetzt werden können.

Die einfachste Variante für einen variabel zu haltenden Dateinamen ist, diesen in einem Text-Field eingeben zu lassen, dieses auszulesen und das Ergebnis anstelle des Dateinamens einzusetzen:

```
FileWriter fw = new FileWriter(jTextField1.getText());
```

Die Gefahr ist jedoch insbesondere beim FileReader groß, dass die gewünschte Datei aufgrund eines Schreibfehlers gar nicht vorhanden ist. Insofern bietet es sich an, lieber gleich auf die aufwendigere, aber sichere Variante, nämlich den JFileChooser, zurückzugreifen.

Ein JFileChooser-Objekt stellt ein Dialog-Fenster für die interaktive Auswahl einer Datei aus dem Verzeichnisbaum zum Lesen bzw. Schreiben bereit. Es kann zwar als Objekt in das Anwendungsfenster gesetzt werden, ist dann aber immer zu sehen. Es sollte also besser als nicht-sichtbares Objekt global eingeführt werden:

```
JFileChooser dateiauswahl = new JFileChooser();
```

Damit eine solche Zeile vom Compiler akzeptiert wird, muss die Swing-Bibliothek mithilfe von „`import javax.swing.*;`“ eingebunden werden, sofern sie nicht schon aufgrund von anderen Swing-Objekten im Quelltext vorhanden ist.

Der Aufruf eines Dialogs zum Öffnen einer Datei erfolgt mithilfe der Methode „`showOpenDialog`“. Diese liefert als Ergebnistyp einen Wert zurück, der angibt, ob der Anwender evtl. „Abbrechen“ angeklickt oder keine Datei ausgewählt hat. In diesem Fall soll natürlich gar nichts passieren. Der gesamte Aufruf des Dialogs und die nachfolgenden Anweisungen werden deshalb in eine `if`-Anweisung eingeschlossen:

```
if (dateiauswahl.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
    FileReader fr = new FileReader(dateiauswahl.getSelectedFile());
    JTextArea1.read(fr, null);
}
```

Die Methode „`getSelectedFile`“ liefert als Ergebnis mit dem Typ „`File`“ die für den `FileReader` notwendige Information über die zu öffnende Datei.

Das Speichern verläuft analog mithilfe der Methode „`showSaveDialog`“:

```
if (dateiauswahl.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
    FileWriter fw = new FileWriter(dateiauswahl.getSelectedFile());
    JTextArea1.write(fw);
}
```

Die nachfolgende Auflistung enthält weitere Einstellungen, die den Umgang mit dem `JFileChooser` etwas komfortabler gestalten:

- ◆ Der `JFileChooser` öffnet in der Regel das Verzeichnis „Eigene Dateien“, sofern nicht explizit ein anderes Verzeichnis angegeben ist. Hierfür steht die Methode „`setCurrentDirectory`“ zur Verfügung. Mithilfe folgender Anweisung wird z. B. das zu öffnende Verzeichnis auf das in der Einführung beschriebene Verzeichnis gesetzt:

```
dateiauswahl.setCurrentDirectory(new File("."));
```

- ◆ Der Öffnen- bzw. Speichern-Dialog kann auch mit einer aussagekräftigen Kopfzeile versehen werden:

```
dateiauswahl.setDialogTitle("Beschriftung der Kopfzeile");
```

- ◆ Für die im Öffnen- bzw. Speichern-Dialog angegebene Datei kann ein Vorschlag unterbreitet werden. Dafür steht die Methode „`setSelectedFile`“ zur Verfügung, die vor der `show`-Anweisung aufgerufen werden muss:

```
dateiauswahl.setSelectedFile(new File("Vorschlag.txt"));
```

Außerdem ist es mithilfe dieser Methode möglich, den beim letzten Aufruf des `JFileChooser`s ausgewählten Dateinamen zu löschen, wenn dieser nicht als Vorschlag erscheinen soll:

```
dateiauswahl.setSelectedFile(new File(""));
```

- ◆ Der Ergebnistyp „`File`“ der Methode „`getSelectedFile`“ stellt mithilfe von „`getName`“ und „`getAbsolutePath`“ zwei Methoden zur Verfügung, die Informationen über den Namen bzw. Speicherort der Datei als `String` liefern können. Um z. B. für das Speichern einer gerade geöffneten Datei einen leicht modifizierten Dateinamen im Speichern-Dialog vorschlagen zu lassen, könnte folgendes Programm verwendet werden:

```
String dateiname = dateiauswahl.getSelectedFile().getName();
int position = dateiname.lastIndexOf('.');
dateiname = dateiname.substring(0,position) + "_neu" +
            dateiname.substring(position);
dateiauswahl.setSelectedFile(new File(dateiname));
```

Eine zum Öffnen angeklickte Datei „`Test.txt`“ würde also für das Speichern als „`Test_neu.txt`“ vorgeschlagen werden. (Dieser Quelltext geht davon aus, dass der User keine Datei ohne Dateierweiterung angeklickt hat.)

- ◆ Der JFileChooser hat die Möglichkeit, die Art der angezeigten Dateien einzuschränken. Hierzu werden FileFilter verwendet, die über folgende Importanweisung eingebunden werden müssen: `import javax.swing.filechooser.FileFilter;`

Ein FileFilter hat immer zwei Methoden: Die Methode „`accept`“ legt fest, welche Dateien und/oder Verzeichnisse angezeigt werden sollen, und die Methode „`getDescription`“ legt die für den User auszuwählende Beschreibung dieser Dateien fest. Der FileFilter wird dem JFileChooser mithilfe der Methode „`addChoosableFileFilter`“ zugeordnet. Dazu sehen wir uns folgendes Beispiel an:

```
dateiauswahl.addChoosableFileFilter(new FileFilter() {
    public boolean accept(File f) {
        if (f.isDirectory()) return true;
        else return f.getName().toLowerCase().endsWith(".txt");
    }
    public String getDescription(){return "Textdateien (*.txt)";}
});
```

Die erste Zeile in der `accept`-Methode sorgt dafür, dass neben den gewünschten Dateien auch Verzeichnisse angezeigt werden, damit man auch durch diese Unterverzeichnisse navigieren kann. Die zweite Zeile wandelt den Dateinamen in Kleinbuchstaben um und prüft, ob dieser mit der angegebenen Datei-Endung „.txt“ endet.

Die Methode „`getDescription`“ sorgt dafür, dass der User in der Format-Auswahl den Text „Textdateien (\*.txt)“ angezeigt bekommt.

Analog kann dem JFileChooser jetzt ein weiterer FileFilter zugeordnet werden, zwischen denen der User auswählen kann. Der zuletzt hinzugefügte FileFilter wird beim Aufruf des JFileChooser angezeigt.

Wenn man die hinzugefügten FileFilter wieder löschen möchte, verwendet man

```
dateiauswahl.resetChoosableFileFilters();
```

Als Standard stellt der JFileChooser immer den Filter „Alle Dateien“ zur Verfügung. Wenn man diesen nicht auswählen können lassen möchte, verwendet man

```
dateiauswahl.setAcceptAllFileFilterUsed(false);
```

## Ausblick: Random-Access-Dateien

Im Gegensatz zu Textdateien bestehen Random-Access-Dateien aus einer Folge von Datensätzen mit festgelegter Länge, z. B. jeweils einem Namen der Länge 10 Zeichen und einer Zahl mit 1 Byte Länge. Da die Länge der Daten jeweils festgelegt ist, ist es möglich, direkt einen bestimmten Datensatz anzusteuern: Der fünfte Datensatz beginnt beim 45. Byte der Datei, weil die vier Datensätze davor  $4 \times 11$  Bytes Platz beanspruchen.

Die Arbeit mit Random-Access-Dateien ist vor allem dann sinnvoll, wenn man im Unterricht den Einsatz von Indizes vorbereiten möchte. Eine ausführliche Darstellung über die Möglichkeiten des Einsatzes findet sich im VLIN-Dokument „Datenstrukturen und Datenbanken - Teil 4b - Dateien mit dem JBuilder“ von Eckart Modrow, das im Material-Archiv des vorangegangenen Kurses zu finden ist.

## Aufgaben

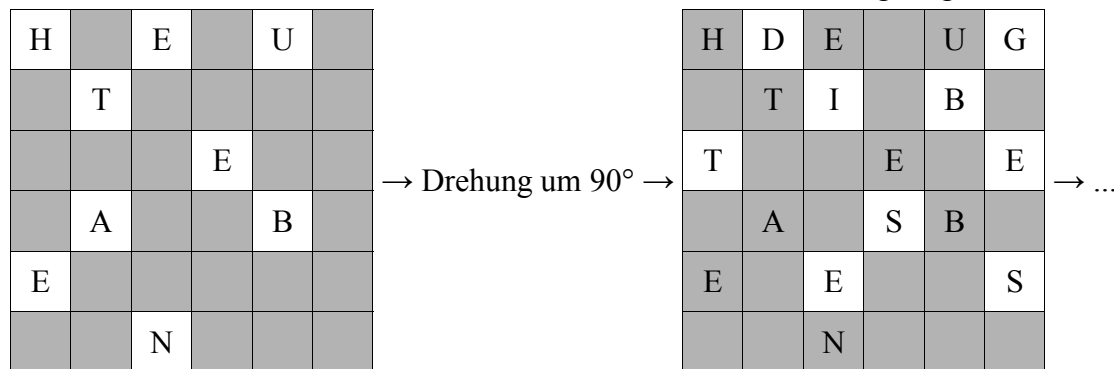
1. Eine Textdatei soll die  $x$ - und  $y$ -Werte der Funktion  $y = x^3 - 2 \cdot x$  im Bereich  $-3 \leq x \leq 2$  enthalten. Wählen Sie eine Schrittweite von 0,1.
2. Ein Programm soll in einer Textdatei alle Leerzeichen durch „X“ ersetzen.
3. Ein Programm soll die Anzahl der Zeilen und Zeichen einer Textdatei ausgeben. Überlegen Sie sich selbst, ob Sie Satzzeichen und Leerzeichen als Zeichen zählen wollen oder nicht. (Hinweise: Wie kann man im Programmablauf für Dateien aller Betriebssysteme ein Zeilenende erkennen? Arbeitet das Programm auch korrekt, wenn in der Datei eine Leerzeile enthalten ist oder wenn die letzte „richtige“ Zeile nicht bzw. doch mit Zeilenende-Zeichen abschließt? Verwenden Sie das Tool „Notepad2“, um verschiedene Zeilenenden zu simulieren.)
4. Suchen Sie im Verzeichnis C:\Windows (und ggf. Unterverzeichnissen) nach einer ausreichend großen ini-Datei. Lassen Sie alle Zeilen ausgeben, die mit „[“ beginnen. (Ausgabe in einer TextArea oder in einer weiteren Datei.)
5. Der häufigste Buchstabe im Deutschen ist das „e“ mit ca. 17,40 %. Ein Programm soll die Häufigkeitsverteilung für deutsche Texte ermitteln.  
Fügen Sie einen Text (z. B. aus dem WWW) in ein TextArea ein und lassen Sie die Anzahl der jeweils gefundenen Buchstaben („ä“ = „ae“ usw.) per Knopfdruck in eine Textdatei schreiben, die pro Zeile einen der 26 Buchstaben und die zugehörige Anzahl enthält. Ein einzelner Text kann noch große Abweichungen gegenüber der „tatsächlichen“ Häufigkeitsverteilung haben. Erweitern Sie deshalb Ihr Programm so, dass nach dem Einfügen eines neuen Textes und Betätigung des Buttons die neu ermittelten Werte zu den vorhandenen in der Textdatei hinzuaddiert werden.
6. CSV-Dateien sind Tabellen, die als Textdatei abgespeichert wurden. Erzeugen Sie eine CSV-Datei mit einem Tabellenkalkulationsprogramm („Speichern unter ...“ und dann CSV-Format auswählen), in der drei Spalten mit Namen ausgefüllt sind (siehe Abbildung).  

	A	B	C
1	Emil	Anton	Julia
2	Fritz	Marie	Merle
3	Birte	Egon	Martin
4	Vanessa	Daniel	Isabell
5			

  
Öffnen Sie die CSV-Datei mit einem Text-Editor und sehen Sie nach, welches Verfahren Ihr Tabellenkalkulationsprogramm verwendet hat, um die Felder zu trennen. (OpenOffice lässt beim Speichern die Wahl. Bei Excel gibt es „Probleme“, wenn ein Name ein Leerzeichen enthält.)  
Schreiben Sie ein Programm, das die zweite Spalte in einer TextArea ausgibt.
7. Eine Textdatei soll in ein HTML-Dokument „übersetzt“ werden. Dabei ist Folgendes zu beachten:
  - ♦ Die Zeilen des HTML-Dokuments sollen nicht länger als 80 Zeichen sein.
  - ♦ Umlaute und „ß“ sollen korrekt in „&uml;“ etc. umgesetzt werden. (Welche weiteren Sonderzeichen müssen noch umgesetzt werden?)
  - ♦ Absätze sollen korrekt in „<p>“ und „</p>“ eingeschachtelt werden.
  - ♦ Die Anführungszeichen der Textdatei, die ja nur als „ “ dargestellt werden, sollen korrekt in typisierte Anführungszeichen – also unten und oben – umgesetzt werden. (Welche Bedingungen müssen eigentlich erfüllt sein, damit ein Anführungszeichen oben sein muss?)
8. Ein Programm soll einen HTML-Quelltext „entschlacken“. Häufig treten folgende Sequenzen auf: „irgendein Text <font = ...></font> weiterer Text“, also ein Tag-Bereich ohne Inhalt. Diese Tags könnten also problemlos gelöscht werden.
9. Aus einem HTML-Dokument sollen alle externen Links in eine weitere Textdatei geschrieben werden. Links werden mithilfe des Tags „<a>“ und dem Parameter „href“ dar-

gestellt. Beispiel: „<a href="http://www.vlin.de">“. Außer dem Parameter „href“ können auch noch weitere Parameter in einem solchen Tag enthalten sein. Wenn der hinter „href=" zugewiesene Wert mit „#“ beginnt, handelt es sich um einen internen Link, der auf eine Stelle innerhalb dieses HTML-Dokuments verweist, also kein externer Link ist. (Hinweis: Suchen Sie beim Einlesen des HTML-Dokuments nach der Zeichenkette „<a“ und speichern Sie alle bis zum ersten Auftreten von „>“ vorkommenden Zeichen in einem anschließend zu untersuchenden String ab.)

10. Der österreichische Oberst Eduard Fleißner von Wostowitz schlug folgende, als „Dreh-raster-Verfahren“ bezeichnete Methode zum Verschlüsseln eines Textes vor: Aus einer quadratischen Schablone mit 36 Feldern werden 9 Felder herausgeschnitten (Bedingungen für diese Felder: siehe unten). Die Schablone wird auf ein Blatt Papier gelegt und der zu verschlüsselnde Text (hier: „Heute Abend gibt es Essen in der Balkansonne.“) wird Buchstabe für Buchstabe in die offenen Felder von links nach rechts und zeilenweise eingetragen. Anschließend wird die Schablone um 90° im Uhrzeigersinn gedreht und die weiteren Buchstaben werden wieder der Reihe nach in die offenen Felder eingetragen:



Nach viermaligem Durchgang ist das Quadrat vollständig ausgefüllt und wird zeilenweise ausgelesen. Der verschlüsselte Text lautet dann:

„HDESUGATILBETNKEIEAANSBNESEODSNENRNB...“

Wenn (wie in diesem Fall) eine Schablone nicht ausreicht, setzt man das Verfahren mit einem weiteren Quadrat fort. Nicht ausgefüllte Felder (am Ende des Textes) werden mit Fantasie-Buchstaben gefüllt.

Welche Bedingung müssen die herausgeschnittenen Felder erfüllen? Offensichtlich dürfen zwei Felder nicht so ausgeschnitten werden, dass sie bei einer Drehung aufeinander fallen würden. Am einfachsten kann dies bewerkstelligt werden, indem man jedem der neun Felder im ersten Quadranten (also oben rechts) den Quadranten zuordnet, auf den dieses Feld der Schablone in der Startposition gedreht sein muss. Die obige Schablone könnte also durch nebenstehende Angabe erzeugt worden sein. (Verwenden Sie diesen Hinweis gleich als „Definition“ für eine Schablone im Programm.)

3	1	2
4	2	3
1	3	2

Ein Programm soll automatisch eine vorhandene Klartext-Textdatei „verschlüsseln“.

Anmerkung: Dieses Verschlüsselungsverfahren ist relativ einfach zu „knacken“; es wurde aber noch im zweiten Weltkrieg im Funkverkehr zwischen deutschen Spionen in Südamerika und der deutschen Abwehr verwendet. Quelle: Kippenhahn, Rudolf: Verschlüsselte Botschaften, rororo 1990, S. 180ff.

11. Hinweise zum Umwandeln von Zahlen in Zahlworte bei der Erstellung eines Klartextes (siehe oben)<sup>1</sup>:

- ◆ Die umzuwandelnde Zahl muss in Vor- und Nachkomma-Anteil zerlegt werden; der Nachkomma-Anteil ist einfach Ziffer für Ziffer zu „übersetzen“.
- ◆ Der Vorkomma-Anteil muss mit führenden Nullen so ergänzt werden, dass die Zeichenanzahl durch 3 teilbar ist. (Tausender-Punkte etc. sind ggf. vorher zu entfernen.)

<sup>1</sup> Diese Aufgabe ist völlig „off-topic“, da sie nichts mit der Verwendung von Dateien zu tun hat.

- ◆ Die Dreiergruppen sind identisch umzuwandeln (siehe nächster Punkt). Je nach Position innerhalb der Zeichenkette wird „tausend“ etc. ergänzt.
- ◆ Innerhalb der Dreiergruppe gibt die erste Stelle die Hunderter an; bei „0“ ist ggf. gar nichts auszugeben.

Die verbleibenden zwei Zeichen werden in eine Zahl umgewandelt. Völlig unproblematisch sind die Zahlen unter 10. Einer Sonderbehandlung bedürfen die Zahlen zwischen 10 und 19 (einschließlich). Alle weiteren Zahlen werden mit der letzten Ziffer zuerst genannt, dann die zweitletzte Ziffer als „...zig“ (bei 4, 5, 8 und 9) und Sonderfälle bei 2, 3, 6 und 7.

Fangen Sie bei der Programmerstellung mit dem letzten Punkt an.