Sprunganweisungen

Es gibt Situationen in denen es notwendig ist, eine Schleife vorzeitig zu beenden. Dazu werden wir uns den Datentyp char noch einmal etwas genauer anschauen.

Java verwendet für den char den sogenannten Unicode-Zeichensatz[1] der zwei Bytes benötigt und damit umfangreicher ist, als der bekannte ASCII-Zeichensatz (American Standard Code for Information Interchange). Er soll einen universellen Zeichensatz für die Darstellung unterschiedlicher Sprachen sein. Die ersten 128 Zeichen entsprechen dabei dem ASCII-Zeichsatz.

Nehmen wir als Ausgangsbeispiel den folgenden Programmabschnitt:

```
for (int i = 33; i < 127; i++) {
  char symbol = (char)i;
  System.out.print(i+": "+symbol+"\t");
  if ((i%8) == 0)
     System.out.println();
}
```

Es werden die ersten sichtbaren Zeichen der ASCII-Tabelle ausgegeben. Mit \t wird ein Tabulatorschritt bei der Ausgabe eingefügt und alle acht Symbole ein Zeilenumbruch vorgenommen, um die Tabelle übersichtlich zu halten.

1) http://www.unicode.org/

ASCII-Tabelle

Wir erhalten auf der Konsole die folgende Ausgabe:

```
33: !   34: "   35: #   36: $   37: %   38: &   39: '   40: (
41: )   42: *   43: +   44: ,   45: -   46: .   47: /   48: 0
49: 1   50: 2   51: 3   52: 4   53: 5   54: 6   55: 7   56: 8
57: 9   58: :   59: ;   60: <   61: =   62: >   63: ?   64: @
65: A   66: B   67: C   68: D   69: E   70: F   71: G   72: H
73: I   74: J   75: K   76: L   77: M   78: N   79: O   80: P
81: Q   82: R   83: S   84: T   85: U   86: V   87: W   88: X
89: Y   90: Z   91: [   92: \   93: ]   94: ^   95: _   96: `
97: a   98: b   99: c   100: d  101: e  102: f  103: g  104: h
105: i  106: j  107: k  108: l  109: m  110: n  111: o  112: p
113: q  114: r  115: s  116: t  117: u  118: v  119: w  120: x
121: y  122: z  123: {  124: |  125: }  126: ~
```

Demzufolge sollten die folgenden Relationen zu welchem Ergebnis führen?

```
boolean a = 'a' < 'b';
boolean b = 'a' < 'A';
```

Sprunganweisungen

Angenommen, wir suchen ein bestimmtes Symbol in der Tabelle und wollen dessen Position ausgeben. Anschließend soll die Schleife beendet werden.

Das Beenden kann beispielsweise dadurch erzwungen werden, indem die Zählvariable einer for-Schleife innerhalb der Schleife auf einen Wert gesetzt wird, der die Bedingung zum Weiterlaufen nicht mehr erfüllt.

Schauen wir uns das Beispiel dazu an:

```
for (int i = 33; i < 127; i++) {
   char symbol = (char)i;
   if (symbol == 'A'){
      System.out.println("Symbol "+symbol+" an Position "+i+" gefunden.");
      i=127;      // schlechter Programmierstil
   } else
      System.out.println("bisher nichts passendes gefunden ...");
}
```

Das ist allerdings sehr unschön und bei komplizierteren Ausdrücken für Leser schlecht nachzuvollziehen, wann die Schleife verlassen wird.

Aus diesem Grund gibt es Sprunganweisungen!

Sprung mit break

Der Sprungbefehl break schließt nicht nur die case-Fälle bei switch, sondern beendet auch unmittelbar while-, do-while- und for-Schleifen:

```
for (int i = 33; i < 127; i++) {
   char symbol = (char)i;
   if (symbol == 'A'){
      System.out.println("Symbol "+symbol+" an Position "+i+" gefunden.");
      break;
   }
   System.out.println("bisher nichts passendes gefunden ...");
}
```

Die for-Schleife wird abgebrochen, wenn das Symbol 'A' identifziert worden ist. Anschließend wird mit den Anweisungen, die nach der Schleife kommen fortgefahren.

Ergänzung zur for-Schleife I

An dieser Stelle wollen wir nochmal kurz auf die Konstruktion einer for-Schleife eingehen. Wir hatten ja angedeutet, dass es auch möglich ist, eine Schleife mit leerem Definitions- und Manipulationsbereich zu definieren:

```
for (;;) {
   // Endlosschleife
}
```

Es gibt keine Bedingung, die unsere Schleife zum Abbruch führen kann. Diese Schleife könnte beispielsweise in den Stellen Anwendung finden, in denen solange etwas auszuführen ist, bis ein bestimmtes Ereignis auftritt.

Dann kann die Schleife mit break beendet werden:

```
for (;;) {
   // tue etwas
   bedingung = prüfen();
   if (bedingung)
      break;
}
```

Wenn die Funktion prüfen ein true liefert, wird die Schleife durch break beendet.

Ergänzung zur for-Schleife II

Besser ist es, in diesem Fall eine while-Schleife zu verwenden und die Schleife abzubrechen, wenn die Bedingung zum Abbrechen erfüllt ist:

```
bedingung = false;
while (!bedingung) {
   // tue etwas
   bedingung = prüfen();
}
```

Dazu müssen wir die Bedingung zu Beginn auf false setzen und bei while negiert überprüfen.

Eleganter ist allerdings diese Lösung:

```
while (!bedingung()) {
   // tue etwas
}
```

Sprung mit continue

Kommen wir zu unserem Ursprungsbeispiel mit der Suche in der ASCII-Tabelle aus dem vorhergehenden Abschnitt zurück.

Jetzt wollen wir alle Großbuchstaben identifizieren und deren Positionen ausgeben:

```
for (int i = 33; i < 127; i++) {
   char symbol = (char)i;
   if ((symbol>='A') && (symbol<='Z'))
      System.out.println("Symbol "+symbol+" an Position "+i+" gefunden.");
   else
      System.out.println("nichts passendes gefunden...");
}
```

Für solche Fälle, in denen wir bei erfolgreicher Suche mit der Schleife weitermachen wollen, können wir das Schlüsselwort continue statt dem if-else-Konstrukt verwenden:

```
for (int i = 33; i < 127; i++) {
   char symbol = (char)i;
   if ((symbol>='A') && (symbol<='Z')){
      System.out.println("Symbol "+symbol+" an Position "+i+" gefunden.");
      continue;
   }
   System.out.println("nichts passendes gefunden...");
}
```

Sprungmarken I

Bei verschachtelten Schleifen wird immer die aktuelle innere Schleife beendet. Um aber explizit anzugeben, zu welcher Schleife gesprungen werden soll, lassen sich Marken unterbringen (markierte Anweisungen).

Diese Marken werden mit folgender Syntax angegeben:

```
<Marke>:
```

Wenn hinter break oder continue eine Marke steht, dann springt das Programm zu der Marke und beendet die Schleife:

```
continue <Marke>;
```

Sprungmarken II

Die Idee besteht darin, Schleifen gezielt beenden zu können oder entsprechend weiterlaufen zu lassen.

Angenommen, wir haben zwei verschachtelte for-Schleifen, die uns die Paare von Groß- und Kleinbuchstaben finden sollen:

```
Aussen:
for (int i = 33; i < 127; i++) {
   for (int j = 33; j < 127; j++) {
      char symbol1 = (char)i;
      char symbol2 = (char)j;
      if (Character.isUpperCase(symbol1) &&
         (symbol2 == Character.toLowerCase(symbol1))){
         System.out.println("Symbol "+symbol1+" ["+
                            i+"] und Symbol "+symbol2+" ["+
                            j+"] sind ein Paar.");
         continue Aussen;
      }
   }
}
```

Wir erhalten:

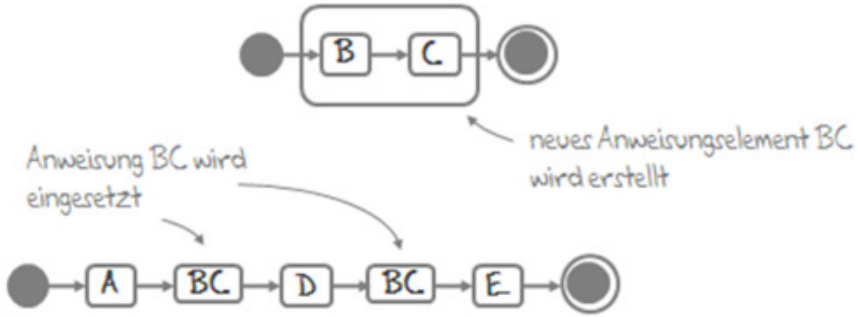
```
Symbol A [65] und Symbol a [97] sind ein Paar.
Symbol B [66] und Symbol b [98] sind ein Paar.
...
Symbol Z [90] und Symbol z [122] sind ein Paar.
```

Ausgelagerte Programmabschnitte in Java

Programme bestehen aus Ketten von Anweisungen, das wissen wir bereits. Oft kommt es dabei vor, dass sich Folgen von Anweisungen wiederholen.

Eine Sequenz von Anweisungen, bei der sich B und C wiederholen.

Wir erstellen eine neue Anweisung mit dem Namen BC, die B und C nacheinander ausführt. Die neue Anweisung BC wird entsprechend eingesetzt:



In diesen Fällen können wir das Programm übersichtlicher gestalten, indem wir diese Anweisungsblöcke zu einer neuen Anweisung, deren Bezeichnung wir fast frei wählen können, zusammenfassen.

Der große Vorteil besteht darin, dass wir den Anweisungsblock nur noch einmal schreiben müssen. Das vermindert die Fehleranfälligkeit.

Motivation zu Funktionen

```
public class Ausgabe{
   public static void main(String[] args){
      int a=4;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable ist "+a);
      System.out.println("*****************************************");
      System.out.println();

      a=(a*13)%12;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable ist "+a);
      System.out.println("*****************************************");
      System.out.println();

      a+=1000;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable ist "+a);
      System.out.println("*****************************************");
      System.out.println();
   }
}
```

Funktionen in Java

Funktionen repräsentieren einen Programmabschnitt, der einmal formuliert beliebig oft aufgerufen und verwendet werden kann. Eine Funktion erhält dabei einen eindeutigen Namen (beginnend mit einem Kleinbuchstaben).

Parameterinhalte können an eine Funktion übergeben werden. Diese gelten zunächst nur innerhalb der Methode (daher werden sie auch lokale Variablen genannt) auch wenn in der main-Methode eine Variable mit dem gleichen Namen existiert, haben diese beiden nichts miteinander zu tun.

Die Syntax unserer Funktionen sieht zunächst erstmal so aus:

```
public static <Datentyp> <Funktionsname>(
    <Datentyp> Parameter1,
    <Datentyp> Parameter2,
     ...) {
     // Funktionskörper
}
```

Aus der Mathematik wissen wir, dass Funktionen auch ein Ergebnis liefern - genau ein Ergebnis. Auch für unsere Funktionen gilt das. Falls, wie in unserem Fall, kein Rückgabewert existiert, schreiben wir als Rückgabewert das Schlüsselwort void.

Ausgabe in Funktion auslagern

Jetzt wollen wir versuchen, mit Hilfe einer Funktion, die Ausgabe ein wenig zu erleichtern. In unserem Ausgabebeispiel ist a ein Eingabeparameter für die neue Funktion.

Die Funktion schreiben wir laut Konvention oberhalb der main-Funktion. Für die Lesbarkeit des Programms sind sprechende Namen, in unserem Beispiel gibAus, unerlässlich:

```
public class AusgabeFunktion{
   public static void gibAus(int a){     // neue Funktion
      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable ist "+a);
      System.out.println("*****************************************");
      System.out.println();
   }

   // main-Funktion
   public static void main(String[] args){
      int a=4;
      gibAus(a);
      a=(a*13)%12;
      gibAus(a);
      a+=1000;
      gibAus(a);
   }
}
```

Funktionen mit Rückgabewert

Schauen wir uns noch ein weiteres Beispiel an. Dazu berechnen wir in einer Funktion für die Eingabe x den Wert f(x)=x*13-1024%(34+12):

```
public class Funktion{
   public static int funktion(int x){
      int wert=(x*13)-1024%(34+12);
      return wert;
   }

   public static void main(String[] args){
      for (int i=0; i<10; i++)
         System.out.println("x="+i+" und f(x)="+funktion(i));
   }
}
```

In diesem Beispiel gibt die Funktion einen int-Wert mit der Anweisung return zurück und wird beendet. Sollten noch Zeilen nach einem return stehen, so gibt Java einen Fehler aus, denn diese Zeilen können nie ausgeführt werden.

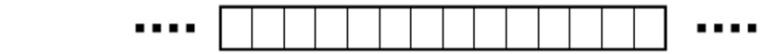
Als Ausgabe erhalten wir:

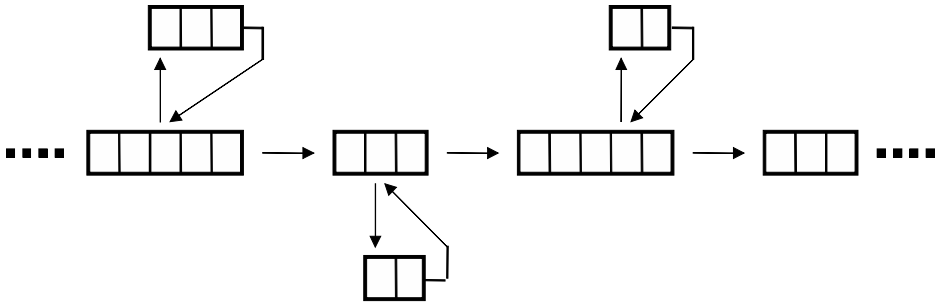
```
C:\Java>java Funktion
x=0 und f(x)=-12
x=1 und f(x)=1
x=2 und f(x)=14
x=3 und f(x)=27
...
x=9 und f(x)=105
```

Wichtiges Konzept

Die Auslagerung von Programmabschnitten in Funktionen ist eines der wichtigsten Programmierkonzepte.

Aus Gründen der Übersichtlichkeit und besseren Fehlersuche werden wir dieses Konzept so oft es geht einsetzen ...





Live-Coding-Session

Vorlesungsteil

Kompromiss zwischen Einsatz und Verständnis

Einer der Hauptgründe für schlechtes Verstehen liegt darin begründet, daß die Leute sich selbst nicht darüber im klaren sind, was sie überhaupt sagen wollen.
Cyril Northcote Parkinson

Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Kompromiss zwischen Einsatz und Verständnis

- Parametrisierter Programmstart
- Daten aus einer Datei lesen
- Daten in eine Datei schreiben
- Daten von der Konsole einlesen
- Kryptographie: Caesar/xor-Codierer
- Projekt: Nachrichtenverschlüsselung
- Projekt: Kleiner Codeknacker

Parametrisierter Programmstart

Wir haben die Möglichkeit, beim Start eines Javaprogramms über die Kommandozeile in der Konsole Parameter zu übergeben:

java <Programmname> <Parameter1> <Parameter2> ...

Das hat den großen Vorteil, dass wir in vielen Fällen, bei denen wir unser Programm beispielsweise testen wollen, nicht jedes Mal in den Programmcode gehen, einen Parameter ändern, die Datei speichern und anschließend kompilieren müssen. Das kann sehr mühsam und zeitaufwendig werden.

Unsere ersten Programme verfügten ja bereits über eine main-Funktion, dessen Signatur uns bei genauerer Betrachtung verrät, dass eine Liste vom Datentyp String übergeben werden kann:

```
public static void main(String[] args) {…}
```

Der Variablenbezeichner args ist dabei willkürlich gewählt, steht aber für solche Fälle zur Verfügung, in denen wir dem Programm einen oder mehrere Parameter vor dem Start mitgeben wollen.

Genau zwei Parameter übergeben

Wir sehen ein Beispiel, bei dem zwei Parameter übergeben werden können:

```
public class MeineEingaben {
   public static void main(String[] args) {
      System.out.println("Eingabe 1: >"+args[0]+"< und");
      System.out.println("Eingabe 2: >"+args[1]+"<");
   }
}
```

Wenn wir zwei Eingaben erwarten, können wir diese mit args[0] (wir sprechen das "args an der Stelle 0" aus) und args[1] einlesen. Der Zählindex beginnt dabei stets bei 0.

Testen wir das Programm, indem wir zwei Eingaben, die durch ein Leerzeichen getrennt sind, an die Kommandozeile zum Aufruf des Programms hängen. Das Programm gibt nun aus, welche zwei Eingaben es erhalten hat:

```
C:\>java MeineEingaben Hallo 27
Eingabe 1: >Hallo< und
Eingabe 2: >27<
```

Zu wenig Parameter

In unserem Beispiel ist es wichtig zu wissen, wie viele Eingaben wir erwarten. Sollten wir auf einen Eintrag in der Stringliste args zugreifen, der keinen Wert erhalten hat, also auch nicht mit der Kommandozeile übergeben wurde, dann passiert das Folgende:

```
C:\>java MeineEingaben Hallo
Eingabe 1: >Hallo< und
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
        at MeineEingaben.main(MeineEingaben:5)
```

Dieses Problem können wir mit folgenden zusätzlichen Zeilen umgehen:

```
public class MeineEingaben {
   public static void main(String[] args) {
      // Anzahl der Eingaben auslesen
      System.out.println(args.length + " Eingaben");

      for (int i=0; i<args.length; i++)
         System.out.println("Eingabe "+i+": >"+args[i]+"<");
   }
}
```

Jetzt ist das Programm ohne Fehler in der Lage, beliebig viele Eingaben auszugeben, ohne es jedes Mal mit javac neu kompilieren zu müssen.

Live-Coding-Session

Daten aus einer Datei lesen

Dem folgenden Programm DateiAuslesen kann ein Dateiname übergeben werden. Die Datei wird dann ausgelesen und der Inhalt auf der Konsole angezeigt:

```
import java.io.*;
import java.util.Scanner;

public class DateiAuslesen {
   public static void main(String[] args){
      // Dateiname wird übergeben
      String dateiName = args[0];

      try {
         // öffne Datei zum Auslesen
         Scanner scanner = new Scanner(new File(dateiName));
         int zaehler = 0;

         // Solange es noch neue Zeilen gibt, lies diese aus
         while (scanner.hasNextLine()) {
            String zeile = scanner.nextLine();
            System.out.println("Zeile "+zaehler+": "+zeile);
            zaehler++;
         }

         // schließe die Datei
         scanner.close();
      } catch(FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

Daten auslesen

Verwenden könnten wir DateiAuslesen beispielsweise mit der Datei namen.dat. Der Inhalt einer Datei kann auf der Konsole mit dem Befehl type angezeigt werden:

```
C:\>type namen.dat
Harald Liebchen
Gustav Peterson
Gunnar Heinze
Paul Freundlich

C:\>java DateiAuslesen namen.dat
Zeile 0: Harald Liebchen
Zeile 1: Gustav Peterson
Zeile 2: Gunnar Heinze
Zeile 3: Paul Freundlich
```

Unser Programm kann eine Datei zeilenweise auslesen und gibt das Eingelesene gleich auf der Konsole aus.

Live-Coding-Session

Daten in eine Datei schreiben

Um Informationen in eine Datei zu speichern, schauen wir uns folgendes Beispielprogramm an:

```
import java.io.*;

public class DateiSchreiben {
   public static void main(String[] args) {
      // Dateiname wird übergeben
      String dateiName = args[0];

      try {
         BufferedWriter myWriter = new BufferedWriter(
            new FileWriter(dateiName, false));

         // schreibe zeilenweise in die Datei dateiName
         myWriter.write("Hans Mueller\n");
         myWriter.write("Gundel Gaukel\n");
         myWriter.write("Fred Feuermacher\n");

         // schliesse die Datei
         myWriter.close();
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: " + eIO);
      }
   }
}
```

Mit der Anweisung write schreiben wir eine Zeichenkette in die durch myWriter bekannte Datei filenameOutput. Mit den zwei Symbolen \n am Ende der Zeilen wird ein Zeilenumbruch getätigt. Das erleichtert anschließend das zeilenweise Auslesen der Datei.

Daten schreiben

Jetzt testen wir unser Programm und verwenden zur Überprüfung das vorher vorgestellte Programm DateiAuslesen:

```
C:\>java DateiSchreiben namen2.dat

C:\>java DateiAuslesen namen2.dat
Zeile 0: Hans Mueller
Zeile 1: Gundel Gaukel
Zeile 2: Fred Feuermacher
```

Wir stellen fest: Es hat funktioniert! Wir haben zunächst die drei Zeilen mit DateiSchreiben in namen2.dat gespeichert und konnten diese anschließend wieder mit DateiAuslesen auslesen.

Live-Coding-Session

Daten von der Konsole einlesen

Jetzt sind wir in der Lage, Parameter beim Programmstart mitzugeben und Daten in Dateien zu speichern und wieder auszulesen, oft ist es aber wünschenswert eine Interaktion zwischen Benutzer und Programm zu haben.

Beispielsweise soll der Benutzer eine Entscheidung treffen oder eine Eingabe machen. Das ist mit der Klasse BufferedReader schnell realisiert:

```
import java.io.*;

public class KonsoleEinlesen {
   public static void main(String[] args) {
      System.out.print("Eingabe: ");

      BufferedReader reader = new BufferedReader(
         new InputStreamReader(System.in));

      try {
         // wir lesen einen String ein
         String zeile = reader.readLine();

         // und geben diesen gleich wieder aus
         System.out.println("Ausgabe: "+zeile);
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: " + eIO);
      }
   }
}
```

Kleiner Test

Wir wollen das Einlesen noch kurz testen:

```
C:\>java Einlesen
Eingabe: Text 1332.44 true
Ausgabe: Text 1332.44 true
```

Live-Coding-Session

Kleiner Ausflug in die Kryptographie

Als praktisches Beispiel werden wir ein kleines Programm schreiben, das Nachrichten mit einem Schlüssel codieren und mit dem gleichen Schlüssel wieder decodieren kann. Mit dem bisherigen Wissen ist das schon sehr leicht umsetzbar.

In der Kryptographie gibt es viele Techniken, die wir an dieser Stelle nicht besprechen wollen. Eine sehr einfache Methode ersetzt dabei Zeichen systematisch durch andere Zeichen (die Klasse dieser Methoden wird auch monoalphabetische Substitution genannt).

Caesar-Codierung I

Bei der Cäsarcodierung werden beispielsweise die Buchstaben innerhalb des Alphabets um eine bestimmte Anzahl verschoben. Diese Anzahl ist auch gerade der Schlüssel zum Text.

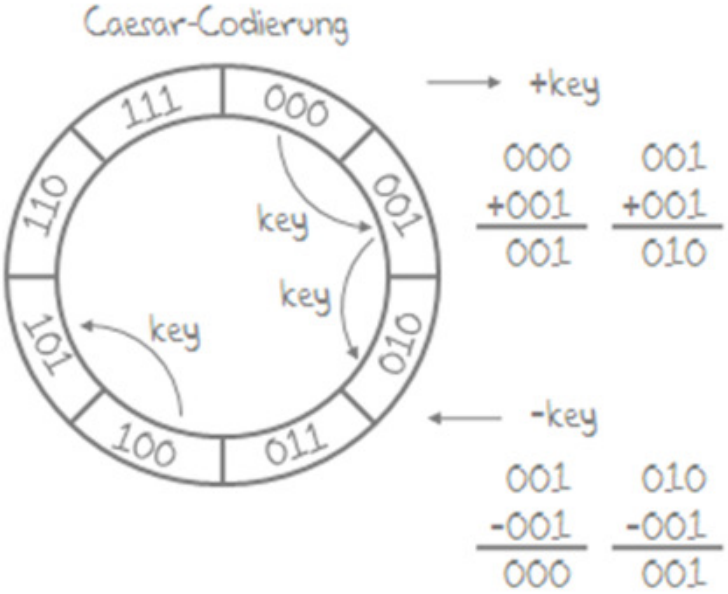
Angenommen wir haben acht Symbole in unserem zu codierenden Alphabet und statt der Symbole verwenden wir deren Binärdarstellung. Für acht Symbole benötigen wir drei Bit. Das Symbol 'A' könnte beispielsweise mit 000, 'B' mit 001, usw. codiert werden. Ein Schlüssel hat jetzt die gleiche Größe von drei Bit und kann beliebig gewählt werden.

Wenn wir jetzt zwei Binärcodierungen addieren, z.B. das Symbol 011 und den Schlüssel 110 (für 6), machen wir das wie bei der schriftlichen Addition:

$$\begin{array}{r} 011 \\ +110 \\ \hline 1001 \end{array}$$

Da uns aber nur drei Bit zur Verfügung stehen, vergessen wir das erste Überlaufbit und erhalten das Symbol 001.

Caesar-Codierung II



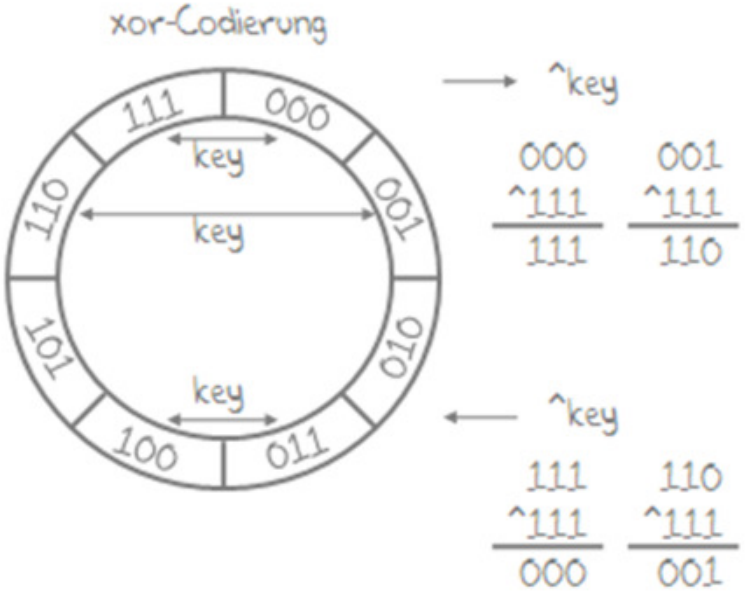
xor-Codierung I

Eine sehr schöne Substitution kann mit der xor-Funktion erzeugt werden, dazu schauen wir uns kurz mal folgendes Beispiel an:

$$\begin{array}{r} 011 \\ \hat{}\,110 \\ \hline 101 \end{array} \qquad \begin{array}{r} 101 \\ \hat{}\,110 \\ \hline 011 \end{array}$$

Auf der linken Seite haben wir das Symbol 011 bitweise mit dem Schlüssel 110 durch die boolesche Funktion xor verknüpft. Rechts ist zusehen, dass der gleiche Schlüssel auf das Resultatsymbol angewendet, das Originalsymbol liefert.

xor-Codierung II



Bitweise logische Operatoren

An dieser Stelle sei kurz erwähnt, dass für die bereits vorgestellten logischen Operationen && und || mit & und | auch bitweise Varianten existieren.

Hier zwei Beispiele:

$$\begin{array}{r} 0110 \\ \&1011 \\ \hline 0010 \end{array} \qquad \begin{array}{r} 0110 \\ |1011 \\ \hline 1111 \end{array}$$

Für logische Ausdrücke sind allerdings && und || vorzuziehen, da die Berechnung frühzeitig beendet werden kann, wenn das Ergebnis bereits feststeht. Dazu ein Beispiel:

```
int zaehler = 1;
int nenner  = 0;

if (nenner!=0 && zaehler/nenner>0)
   System.out.println("Division erlaubt!");
```

Projekt: Nachrichtenverschlüsselung I

Schauen wir uns zunächst die codiere-Funktion an:

```
public static String codiere(String text, int key) {
    // wir werden die Zeichen individuell codieren
    char[] zeichen = text.toCharArray();

    // bitweise xor-Verschlüsselung
    for (int i=0; i<zeichen.length; i++)
       zeichen[i] = (char)(zeichen[i]^key);

    // wir erzeugen aus dem Array vom Typ char einen String
    return new String(zeichen);
 }
```

Projekt: Nachrichtenverschlüsselung II

```
import java.io.*;
import java.util.Scanner;

public class Codierer {
   public static String codiere(String text, int key) {
      ...
   }

   public static void main(String[] args) {
      String dateiIn  = args[0]; // zu lesende Datei
      String dateiAus = args[1]; // zu speichernde Datei

      // geheimer Schlüssel wird in eine Zahl umgewandelt
      int schluessel  = Integer.parseInt(args[2]);

      try {
         // notwendige Programme zum Ein- und Auslesen
         Scanner scanner         = new Scanner(new File(dateiIn));
         BufferedWriter myWriter = new BufferedWriter(new FileWriter(dateiAus, false));

         while (scanner.hasNextLine()) {
            myWriter.write(codiere(scanner.nextLine(), schluessel)+"\n");
         }

         myWriter.close();
         scanner.close();
      } catch(IOException eIO) {
        eIO.printStackTrace();
      }
   }
}
```

Geheimes ...

Nehmen wir an, dass wir eine Datei NachrichtOriginal.txt im gleichen Ordner vorliegen haben und wollen diese jetzt mit dem Schlüssel 41 codieren und anschließend wieder decodieren:

```
C:\>java Codierer NachrichtOriginal.txt CodierterText.txt 41

C:\>java Codierer CodierterText.txt NachrichtEntschluesselt.txt 41
```

Live-Coding-Session

Einfache Codes knacken

$$\begin{array}{rr} 011 & symbol \\ \hat{}\,110 & \hat{}\,schluessel \\ \hline 101 & symbol\ codiert \end{array} \qquad \begin{array}{rr} 011 & symbol \\ \hat{}\,101 & \hat{}\,symbol\ codiert \\ \hline 110 & schluessel \end{array}$$

```
String text = "Das vorliegende Buch ist ein Einsteigerbuch " +
              "und für jeden geeignet, der das Programmieren mit " +
              "der Sprache Java erlernen möchte. Es ist Teil des " +
              "didaktischen Konzepts, einige Hinweise doppelt " +
              "oder sogar dreifach zu erwähnen, wenn die Wieder" +
              "holung dem besseren Verständnis dient. Wir werden " +
              "in vielen Programmübungen gemeinsam Fehler machen " +
              "und diese dann analysieren und korrigieren.";

// Schlüssel wird zufällig ermittelt
int schluessel           = (int)(Math.random()*100) + 1;
String textCodiert       = codiere(text, schluessel);

char maxi                = haeufigstesSymbol(textCodiert);
int schluesselDecodiert  = maxi ^ 'e';
String textDecodiert     = codiere(textCodiert, schluesselDecodiert);
```

Häufigstes Symbol

```
public static char haeufigstesSymbol(String text) {
   char[] symbole      = text.toCharArray();
   int[] symbolCounter = new int[symbole.length];

   int maxCount = 0;
   int maxIndex = 0;
   for (int i=0; i<symbole.length; i++) {
      for (int j=i; j<symbole.length; j++)
         if (symbole[i]==symbole[j])
            symbolCounter[i]++;
         if (maxCount<symbolCounter[i]) {
            maxCount = symbolCounter[i];
            maxIndex = i;
         }
      }

   return symbole[maxIndex];
}
```

Vorlesungsteil

Verwendung einfacher Datenstrukturen

Was nicht auf einer einzigen Manuskriptseite zusammengefasst werden kann,
ist weder durchdacht noch entscheidungsreif.
Dwight D. Eisenhower

Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Verwendung einfacher Datenstrukturen

- Arrays
- Indizierung und typische Fehler
- literale Erzeugung
- elementeweises Kopieren versus clone
- Performancevergleiche selbst vornehmen
- vereinfachte for-Schleife
- Matrizen und mehrdimensionale Arrays
- Conway's Game of Life
- JavaGotchi
- Zustandsdiagramme
- Enumeratoren

Der Begriff Datenstruktur

Das Wort Datenstruktur verrät schon, dass es sich um Daten handelt, die in irgendeiner Weise in Strukturen, die spezielle Eigenschaften besitzen, zusammengefasst werden.

Diese Eigenschaften können sich z.B. darin auswirken, dass ein bestimmtes Datum (an dieser Stelle ist nicht das zeitliche Datum, sondern der Singular von Daten gemeint) schneller gefunden oder eine große Datenmenge platzsparender gespeichert werden kann.

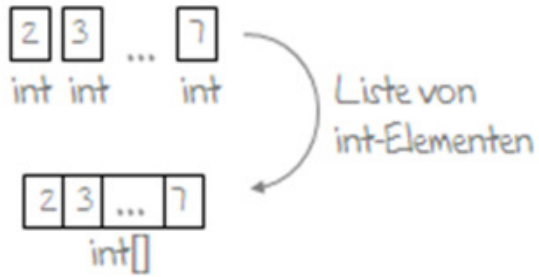
Motivation für Arrays

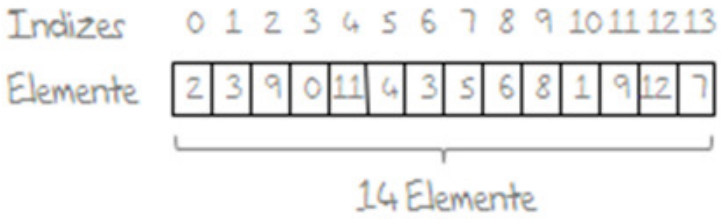
Nehmen wir an, wir möchten nicht nur einen int, sondern viele davon verwalten. Dann könnten wir dies, mit dem uns bereits bekannten Wissen, in etwa so bewerkstelligen:

```
int a, b, c, d, e, f;
a=0;
b=1;
c=2;
d=3;
e=4;
f=5;
```

Das ist sehr aufwändig und nicht besonders elegant. Einfacher wäre es, wenn wir sagen könnten, dass wir n verschiedene int-Werte speichern möchten und diese dann über ihre Position innerhalb von n (also einen Index) ansprechen.

Genau das nennen wir eine Liste oder ein Array von Elementen:





Achtung bei Indizierung der Arrayelemente

Zu beachten ist, dass das erste Element eines Arrays mit dem Index 0 und das letzte der n Elemente mit dem Index n-1 angesprochen werden.

Daran müssen wir uns gewöhnen und es ist eine beliebte Fehlerquelle. Es könnte sonst passieren, dass wir z.B. in einer Schleife alle Elemente durchlaufen möchten, auf das n-te Element zugreifen und einen Fehler verursachen:

```
int[] a = new int[10]; // Erzeugung eines Arrays der Größe 10
for (int i=0; i<=10; i++)
   System.out.println("a["+i+"]="+a[i]);
```

Warum ergibt das einen Fehler?

Erzeugter Fehler durch falschen Indexzugriff

Wenn wir diesen Programmabschnitt aufrufen, erhalten wir folgende Ausgabe mit Fehler:

```
C:\Java>java Array
a[0]=0
a[1]=0
a[2]=0
a[3]=0
a[4]=0
a[5]=0
a[6]=0
a[7]=0
a[8]=0
a[9]=0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
10 at Array.main(Array:5)
```

Wieder erhalten wir die bekannte ArrayIndexOutOfBoundsException. Sie tritt immer dann auf, wenn wir auf den Index eines Arrays zugreifen, den es gar nicht gibt.

Glücklicherweise liefert Java an dieser Stelle einen Fehler. Das ist bei anderen Programmiersprachen, wie z.B. C++, nicht immer der Fall.

Kleiner Rückblick zur main-Methode

Im letzten Teil wurde gezeigt, wie Daten an ein Programm bei dessen Aufruf zu übergeben sind. Hier sehen wir nochmal den Codeabschnitt:

```
public static void main(String[] args){
   for (int i=0; i<args.length; i++)
      System.out.println("Eingabe "+i+": >"+args[i]+"<");
}
```

Auch hier wurde Gebrauch von einem Array args gemacht, in diesem Fall ein Array von Strings. Die Variablenbezeichnung args wird an dieser Stelle sehr häufig verwendet, ist aber beliebig und kann geändert werden.

Deklaration und Zuweisung

Um ein n-elementiges Array zu erzeugen, können wir das Array zunächst deklarieren und anschließend mit dem Befehl new den notwendigen Speicherplatz dafür bereitstellen:

```
<Datentyp>[] <name>;
<name> = new <Datentyp>[n];
```

Beide Zeilen lassen sich auch zu einer zusammengefassen:

```
<Datentyp>[] <name> = new <Datentyp>[n];
```

Literale Erzeugung

Es wurde bereits ein Beispiel gezeigt, wie über einen Index auf die einzelnen Elemente zugegriffen werden kann. Wir haben auch gesehen, dass die Einträge eines int-Arrays mit 0 initialisiert wurden.

Die Initialisierung findet aber nicht bei jedem Datentypen und nicht in jeder Programmiersprache statt, daher wird allgemein empfohlen, in einer Schleife dafür Sorge zutragen.

In dem folgenden Beispiel wollen wir ein Array erzeugen und neue Daten hineinschreiben:

```
int[] a = new int[2];
a[0]    = 3;
a[1]    = 4;
```

Sollten wir schon bei der Erzeugung des Arrays wissen, welchen Inhalt die Elemente haben sollen, dann können wir das mit der sogenannten literalen Erzeugung vornehmen:

```
int[] a = {1,2,3,4,5};
```

Arraygröße bleibt unverändert

Wichtig an dieser Stelle ist noch zu wissen, dass wir die Größe eines erzeugten Arrays nachträglich nicht mehr ändern können. Sollten wir mehr Platz benötigen, bleibt uns keine andere Lösung, als ein größeres Array zu erzeugen und die Elemente zu kopieren.

Wir haben beispielsweise ein Array mit 10 Elementen und wollen ein neues Array mit 20 Elementen erzeugen und anschließend die 10 Elemente kopieren:

```
char[] textListe1 = new char[10];

char[] textListe2 = new char[20];
for (int i=0; i<textListe1.length; i++)
   textListe2[i] = textListe1[i];
```

Elementeweises Kopieren versus clone

Für das Kopieren eines Arrays stehen uns verschiedene Möglichkeiten zur Verfügung. Einen Weg haben wir im vorhergehenden Beispiel gesehen: das elementeweise Kopieren.

Interessanterweise gibt es keine schnellere Alternative. Wer das Kopieren in einer Zeile vornehmen möchte, kann für ein Array liste beispielsweise auf die clone-Funktion zurückgreifen:

<Datentyp>[] <name> = (<Datentyp> [])liste.clone();

Für einen kurzen Performancevergleich der beiden Methoden gibt es in Java mit System.currentTimeMillis() beispielsweise die Möglichkeit, die aktuelle Systemzeit in Millisekunden vor und nach einem Programmabschnitt auszulesen und anschließend die Differenz anzugeben:

```
long startZeit = System.currentTimeMillis();

// Programmabschnitt, dessen Zeit gemessen werden soll

long stopZeit  = System.currentTimeMillis();
long differenz = stopZeit-startZeit;       // in ms
```

Liste für Performancevergleich vorbereiten

In einem der späteren Kapitel werden wir nocheinmal intensiv auf das Thema Zufallszahlen zu sprechen kommen. Hier genügt es zu wissen, dass wir eine Funktion Math.random verwenden, die Zufallszahlen vom Typ double gleichverteilt im Bereich von [0,1) liefert.

Wenn wir also die Zufallszahl mit 100 multiplizieren und in einen int umwandeln, erhalten wir eine Zufallszahl aus dem Intervall [0,99].

Wir können uns für unseren Vergleich der beiden Kopiermethoden eine Liste mit 1000 Elementen zufällig füllen:

```
int[] liste = new int[1000];
for (int i=0; i<liste.length; i++)
   liste[i] = (int)(100*Math.random());
```