Technische Universität München
Fakultät für Informatik
C. Harlfinger, A. Harrer

WS 1998/99
Lösungsvorschläge zu Blatt 1
26. Oktober 1998

Übungen zum Programmieren in Java

Aufgabe 1 Auswertung von Ausdrücken

Gegeben seien folgende Ausdrücke:

(1) `10 == 1 + a ++ * 3`

(2) `a <= b & a > b && true`

(3) `a < b / a ? a *= a : b`

a) Die Ausdrücke in vollständiger Klammerung:

(1) `10 == (1 + ((a ++) * 3))`

(2) `((a <= b) & (a > b)) && true`

(3) `(a < (b / a)) ? (a *= a) : b`

b) Die Werte der Ausdrücke:

(1) `10 == (1 + ((a ++) * 3))`

```
(i) 10 == (1 + ((a ++) * 3))     =
    10 == (1 + (2 * 3))          =
    10 == (1 + 6)                =
    10 == 7                      = false

(ii) 10 == (1 + ((a ++) * 3))    =
     10 == (1 + (3 * 3))         =
     10 == (1 + 9)               =
     10 == 10                    = true
```

(2) Für beide Variablenbelegungen (i) und (ii) gilt:

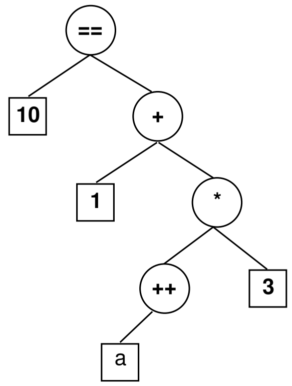
```
((a <= b) & (a > b)) && true    =
false && true                   = false
```

(3) `(a < (b / a)) ? (a *= a) : b`

```
(i) (2 < (6 / 2)) ? (a *= a) : b    =
    (2 < 3) ? (a *= a) : b          =
    true ? (a *= a) : b             =
    (a *= a)                        =
    (2 * 2)                         = 4
```

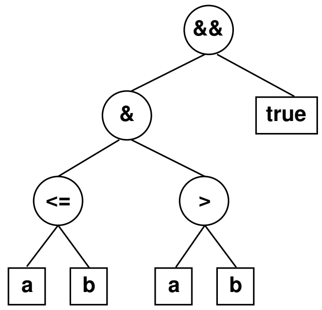
(ii)
```
(3 < (10 / 3)) ? (a *= a) : b   =
(3 < 3) ? (a *= a) : b          =
false ? (a *= a) : b            =
b                               = 10
```

c) Operatorbäume: (1)

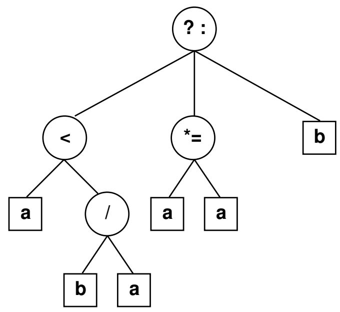


Bemerkung: In diesem Baum wurde der Post-Inkrement-Operator dadurch gekennzeichnet, daß das *a*, auf das der Operator angewendet wird, nach links verzweigt dargestellt wird (das *a* vor dem `++`). Bei einem Prä-Inkrement könnte man entsprechend nach rechts antragen.

(2)



(3)



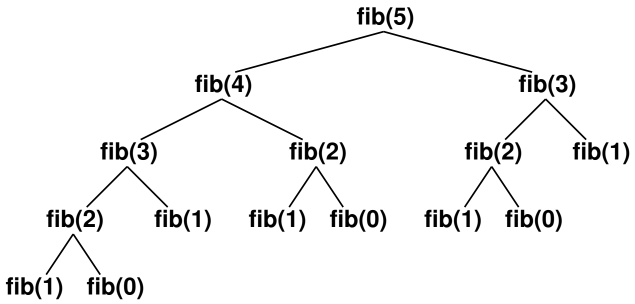
Aufgabe 2 rekursive Programmierung

a) rekursive Methodendefinition zur Berechnung der n-ten Fibonaccizahl:

```
static int fibRekursiv(int n) {

  if (n==0) return 0;
  else if (n==1) return 1;
  else return fibRekursiv(n-1) + fibRekursiv(n-2);
}
```

b) Aufrufbaum für fibRekursiv(5):



Aufgabe 3 iterative Programmierung

a) In einer iterativen Fassung zur Berechnung von Fibonacci-Zahlen ist es sinnvoll, die zuletzt errechneten Zahlen in Hilfsvariablen zu halten und die nachfolgende Zahl aus den beiden zuletzt errechneten Zahlen zu gewinnen.

Das führt zur folgenden iterativen Methodendeklaration zur Berechnung der n-ten Fibonacci-Zahl:

```
static int fibIterativ(int n) {
  if (n==0) return 0;
  else if (n==1) return 1;

  // Fuer n >= 2:
  int vorletztesFib = 0;      //Variable für vorletzten Wert fib(i-2)
                              //Vorbelegung: fib(0)
  int letztesFib = 1;         //Variable für letzten Wert fib(i-1)
                              //Vorbelegung: fib(1)
  int fib = 0;                //Variable für aktuellen Wert fib(i)
                              //Vorbelegung: egal
  int i = 1;                  //Zaehler
  while (i < n) {
    fib = letztesFib + vorletztesFib;
    vorletztesFib = letztesFib;
    letztesFib = fib;
    i++;
  }
  return fib;
}
```

Im Vergleich zur rekursiven Version wird jeder Wert nur einmal berechnet, nicht wie im Aufrufbaum der vorigen Aufgabe ersichtlich evtl. mehrmals.

b) Testausgabe der rekursiven und iterativen Berechnung:

```
public class Fib {

  static int fibRekursiv(int n) {...} // siehe Aufgabe 2a

  static int fibIterativ(int n) {...} // siehe Aufgabe 3a
```

```
  public static void main(String[] args) {
    for (int i = 0; i<= 20; i++)
      System.out.println(i + "-te Zahl\t" + fibRekursiv(i) +
                         "\t" + fibIterativ(i));
  }
}
```

Bei der Ausführung erhalten wir jeweils die Zahlenfolge

`0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765`

Der vollständige Programmtext ist in der Datei

`/usr/proj/javakurs/Tag1/loesungen/Fib.java`

abgelegt.

Aufgabe 4 rekursiver Sortier-Algorithmus

Das vorgeschlagene Sortierverfahren entspricht prinzipiell dem bekannten *Quicksort*-Algorithmus.
Das Grundprinzip dieses Algorithmus ist die Zerlegung des zu lösenden Gesamtproblems (die Sortierung der Werte im Feld `feld`) in mehrere kleinere Teilprobleme, die jeweils leichter zu lösen sind als das Gesamtproblem. Die Lösungen dieser Teilprobleme werden dann zusammengesetzt zur Gesamtlösung. Algorithmen, die nach diesem Prinzip arbeiten, werden oft als *divide-and-conquer*-Verfahren bezeichnet.

a) Vor einer Sortierung muß zunächst einmal überprüft werden, ob `feld` überhaupt sinnvoll zerlegt werden kann. Ist `feld` leer (nicht initialisiert) oder enthält nur ein Element, dann ist keine sinnvolle Zerlegung mehr möglich. In diesem Fall ist aber auch keine Sortierung mehr notwendig, d.h. der Algorithmus ist bereits (erfolgreich) abgeschlossen. Nur wenn `feld` mehr als ein Element enthält, ist eine weitere Zerlegung nötig.

Die Nicht-Zerlegbarkeit von `feld` ist die *Terminierungsbedingung* des rekursiven Sortier-Algorithmus. Sobald diese Bedingung erfüllt ist, sind keine weiteren rekursiven Aufrufe mehr erforderlich.

Damit ergibt sich folgendes Ablaufschema für die rekursive Methode `sortiere`:

1. Überprüfe, ob das Feld `feld` noch weiter zerlegbar ist:
   Wenn nicht, ist die Ausführung von `sortiere` beendet.

2. Zerlege das Feld `feld` in die Teilfelder `kleinerFeld`, `gleichFeld` und `groesserFeld`.

3. Sortiere die Teilfelder `kleinerFeld` und `groesserFeld` jeweils rekursiv (nach demselben Schema).

4. Kopiere die sortierten Teilfelder `kleinerFeld`, `gleichFeld` und `groesserFeld` hintereinander in das Feld `feld`.

5. Beende die Ausführung von `sortiere`.

1. Schritt (Terminierungsbedingung):

Die Erfüllung der Terminierungsbedingung kann direkt anhand der Feldlänge des formalen Parameters `feld` überprüft werden:

```
// Ueberpruefung der Terminierungsbedingung
if ((feld == null) ||         // feld ist leer (nicht initialisiert)
    (feld.length <= 1))       // feld enthaelt hoechstens 1 Element
                              // (beachte: nicht-strikte Auswertung!)
  return;                     // beende Methodenausfuehrung
                              // (keine Zerlegung mehr notwendig)
```

2. Schritt (Zerlegung):

Zunächst stellt sich die Frage nach der Auswahl der Schnittzahl, die die Zerlegungsgrenzen festlegt. Hier sind verschiedene Auswahlverfahren denkbar, die abhängig von der Werteverteilung im zu sortierenden Feld die *Effizienz* des Algorithmus stark beeinflussen können. Wir entscheiden uns für eine einfache Lösung und nehmen einfach den Wert des 1. Feldelements als Schnittzahl:

```
// Festlegung der Schnittzahl
int schnittzahl = feld[0];  // Schnittzahl: Wert des 1. Feldelements
```

Die Zerlegung des Feldes `feld` erzeugt drei Teilfelder. Da `feld` Elemente des Datentyps `int` enthält, deklarieren wir auch unsere Teilfelder dementsprechend, wobei wir zu Beginn von leeren Feldern ausgehen und die Teilfelder daher mit `null` initialisieren:

```
// Deklaration der Teilfelder
int[] kleinerFeld  = null;
int[] gleichFeld   = null;
int[] groesserFeld = null;
```

Die Initialisierung dieser Felder muß so erfolgen, daß jedes der Felder groß genug ist, um alle Werte, die ihm zugewiesen werden, aufnehmen zu können. Um nicht unnötig Speicherplatz zu verbrauchen, legen wir die Feldlängen genau so fest, daß sie den Anzahlen der aufzunehmenden Werte entsprechen.

Da Felder in Java nach ihrer Initialisierung eine feste, nicht mehr veränderbare Länge haben, müssen die Längen der einzelnen Felder vor der eigentlichen Zerlegung festgelegt werden. Zur Ermittlung dieser Werte ist ein kompletter Durchlauf durch `feld` notwendig, bei dem jedes Element von `feld` mit der Schnittzahl verglichen werden muß, um festzustellen, in welches Teilfeld dieses Element später übertragen werden muß:

```
// Berechnung der Teilfeldlaengen
int kleinerAnzahl = 0;
int gleichAnzahl = 0;
int groesserAnzahl = 0;
for (int i = 0; i < feld.length; i++) {
  if (feld[i] < schnittzahl)
    kleinerAnzahl++;
  else
```

```
      if (feld[i] == schnittzahl)
        gleichAnzahl++;
      else
        groesserAnzahl++;
  }
```

Bei der Feldinitialisierung muß berücksichtigt werden, daß eine Initialisierung von 0 Feldelementen (`new int[0]`) in Java nicht zulässig ist. Daher muß überprüft werden, ob die Teilfelder überhaupt Elemente enthalten sollen. Diese Überprüfung ist für das Feld `gleichFeld` nicht nötig, weil hier immer mindestens ein Element (das Schnittzahl-Element, also `feld[0]`) eingefügt werden muß:

```
  // Initialisierung der Teilfelder
  if (kleinerAnzahl > 0)
    kleinerFeld  = new int[kleinerAnzahl];
  gleichFeld   = new int[gleichAnzahl];
  if (groesserAnzahl > 0)
    groesserFeld = new int[groesserAnzahl];
```

Die eigentliche Zerlegung läuft ähnlich ab wie die Längenermittlung. Allerdings werden hier die Indexpositionen für die einzelnen Felder anstatt der Feldlängen inkrementiert, um keine bereits belegten Feldelemente zu überschreiben. Außerdem wird natürlich jedes Element von `feld` entsprechend dem Resultat des Vergleichs mit der Schnittzahl in sein Teilfeld kopiert:

```
  // Zerlegung von feld
  int kleinerIndex = 0;
  int gleichIndex = 0;
  int groesserIndex = 0;
  for (int i = 0; i < feld.length; i++) {
    if (feld[i] < schnittzahl)
      kleinerFeld[kleinerIndex++] = feld[i];
    else
      if (feld[i] == schnittzahl)
        gleichFeld[gleichIndex++] = feld[i];
      else
        groesserFeld[groesserIndex++] = feld[i];
  }
```

<u>3. Schritt (Sortierung der Teilfelder):</u>
Durch die Zerlegung im 1. Schritt können wir nun sicher sein, daß alle Elemente im Feld `kleinerFeld` echt kleiner sind als jedes Element in `gleichFeld`. Ebenso sind die Elemente in `groesserFeld` echt größer als die Elemente der anderen Teilfelder. Die Zerlegung ist somit auf alle Fälle *disjunkt*, d.h. die Wertebereiche der Teilfelder überschneiden sich nicht.

Allerdings müssen die einzelnen Teilfelder noch sortiert werden. Dazu werden zwei rekursive Aufrufe benötigt:

```
  // Rekursive Aufrufe zur Sortierung der Teilfelder
  sortiere(kleinerFeld);
  sortiere(groesserFeld);
```

Für `gleichFeld` ist eine Sortierung unnötig, da alle Elemente dieses Teilfelds denselben Wert besitzen, d.h. dieses Teilfeld ist bereits sortiert.

4. Schritt (Kopieren der Teilfelder):
Stellen wir uns die Felder von links nach rechts aufsteigend indiziert vor. Dann sollen alle Elemente in `kleinerFeld` nach Abschluß der Sortierung links (bei aufsteigender Sortierung) von jedem Element in `gleichFeld` stehen, die ihrerseits wieder links von jedem Element in `groesserFeld` stehen sollen. Wir müssen also der Reihe nach die einzelnen sortierten Teilfelder in das Feld `feld` zurückschreiben:

```
// Zusammenfuegen des Ergebnisses
int i = 0;
for (int j = 0; j < kleinerAnzahl; j++)
  feld[i++] = kleinerFeld[j];
for (int j = 0; j < gleichAnzahl; j++)    // i nicht zuruecksetzen!
  feld[i++] = gleichFeld[j];
for (int j = 0; j < groesserAnzahl; j++)  // i nicht zuruecksetzen!
  feld[i++] = groesserFeld[j];
```

Der 5. Schritt, also das Beenden der Methodenausführung, erfordert in diesem Fall keine eigenen Anweisungen, da kein Rückgabewert angegeben werden muß. Die Ausführung wird automatisch durch das Erreichen des Endes des Methodenrumpfs beendet.

Als Gesamtlösung erhalten wir damit:

```
static void sortiere (int[] feld) {
  // 1. Schritt: Ueberpruefung der Terminierungsbedingung
  if ((feld == null) || (feld.length <= 1))  // nicht-strikte Auswertung
    return;                                 // beende Methodenausfuehrung

  // 2. Schritt: Zerlegung des Feldes
  // Festlegung der Schnittzahl
  int schnittzahl = feld[0];    // Schnittzahl: Wert des 1. Feldelements

  // Deklaration der Teilfelder
  int[] kleinerFeld  = null;
  int[] gleichFeld   = null;
  int[] groesserFeld = null;

  // Berechnung der Teilfeldlaengen
  int kleinerAnzahl = 0;
  int gleichAnzahl = 0;
  int groesserAnzahl = 0;
  for (int i = 0; i < feld.length; i++) {
    if (feld[i] < schnittzahl)
      kleinerAnzahl++;
    else
      if (feld[i] == schnittzahl)
        gleichAnzahl++;
      else
        groesserAnzahl++;
```

```
  }

  // Initialisierung der Teilfelder
  if (kleinerAnzahl > 0)
    kleinerFeld  = new int[kleinerAnzahl];
  gleichFeld   = new int[gleichAnzahl];
  if (groesserAnzahl > 0)
    groesserFeld = new int[groesserAnzahl];

  // Zerlegung von feld
  int kleinerIndex = 0;
  int gleichIndex = 0;
  int groesserIndex = 0;
  for (int i = 0; i < feld.length; i++) {
    if (feld[i] < schnittzahl)
      kleinerFeld[kleinerIndex++] = feld[i];
    else
      if (feld[i] == schnittzahl)
        gleichFeld[gleichIndex++] = feld[i];
      else
        groesserFeld[groesserIndex++] = feld[i];
  }

  // 3. Schritt: Rekursive Aufrufe zur Sortierung der Teilfelder
  sortiere(kleinerFeld);
  sortiere(groesserFeld);

  // 4. Schritt: Zusammenfuegen des Ergebnisses
  int i = 0;
  for (int j = 0; j < kleinerAnzahl; j++)
    feld[i++] = kleinerFeld[j];
  for (int j = 0; j < gleichAnzahl; j++)    // i nicht zuruecksetzen!
    feld[i++] = gleichFeld[j];
  for (int j = 0; j < groesserAnzahl; j++)  // i nicht zuruecksetzen!
    feld[i++] = groesserFeld[j];

  // 5. Schritt: Aufruf beenden (Ende des Methodenrumpfs)
}
```

b) Für die geänderte Methode wird nun ein Rückgabewert erwartet, und zwar ein Feld mit den Eingabewerten in aufsteigender Sortierung. Die entsprechend abgewandelte Methodensignatur sieht damit so aus:

```
static int[] sortiere (int[] feld)
```

Wir deklarieren zunächst ein Feld für das Ergebnis, daß entsprechend der Länge des Eingabefelds initialisiert wird:

```
// Deklaration des Ausgabefelds
int[] ergebnis = null;
```

```
if (feld != null)
  ergebnis = new int[feld.length];
```

Außerdem muß die Überprüfung der Terminierungsbedingung geändert werden. Anstatt einfach die Methodenausführung zu beenden, wird nun `ergebnis` zurückgegeben. Falls `feld` ein Element enthält, so muß dieses zuvor noch kopiert werden (eine Sortierung ist natürlich nicht nötig):

```
// Ueberpruefung der Terminierungsbedingung
if ((feld == null) || (feld.length <= 1)) {// nicht-strikte Auswertung
  if (feld != null)                        // falls feld nicht leer ist,
    ergebnis[0] = feld[0];                 // kopiere einziges Element
  return ergebnis;                         // liefere Ergebnis zurueck
}
```

Die weiteren Änderungen gegenüber der in Teilaufgabe **a)** deklarierten Methode sind in den Schritten 3, 4 und 5 vorzunehmen. Zum einen müssen die prozeduralen Aufrufe der Methode `sortiere` zu funktionalen Aufrufen umgebaut werden:

```
// Rekursive Aufrufe zur Sortierung der Teilfelder
kleinerFeld  = sortiere(kleinerFeld);
groesserFeld = sortiere(groesserFeld);
```

Die Zusammenstellung des sortierten Ergebnisfelds verläuft analog zur Lösung in **a)**. Das Ziel der Kopieraktion ist nun allerdings das Feld `ergebnis`, das anschließend noch explizit zurückgeliefert werden muß:

```
// Zusammenfuegen des Ergebnisses
int i = 0;
for (int j = 0; j < kleinerAnzahl; j++)
  ergebnis[i++] = kleinerFeld[j];
for (int j = 0; j < gleichAnzahl; j++)    // i nicht zuruecksetzen!
  ergebnis[i++] = gleichFeld[j];
for (int j = 0; j < groesserAnzahl; j++)  // i nicht zuruecksetzen!
  ergebnis[i++] = groesserFeld[j];
return ergebnis;
```

Die komplette Lösung lautet damit wie folgt:

```
static int[] sortiere (int[] feld) {
  // Deklaration des Ausgabefelds
  int[] ergebnis = null;
  if (feld != null)
    ergebnis = new int[feld.length];

  // 1. Schritt: Ueberpruefung der Terminierungsbedingung
  if ((feld == null) || (feld.length <= 1)) {// nicht-strikte Auswertung
    if (feld != null)                        // falls feld nicht leer ist,
```

```
      ergebnis[0] = feld[0];               // kopiere einziges Element
    return ergebnis;                       // liefere Ergebnis zurueck
  }

  // 2. Schritt: Zerlegung des Feldes
  // Festlegung der Schnittzahl
  int schnittzahl = feld[0];    // Schnittzahl: Wert des 1. Feldelements

  // Deklaration der Teilfelder
  int[] kleinerFeld  = null;
  int[] gleichFeld   = null;
  int[] groesserFeld = null;

  // Berechnung der Teilfeldlaengen
  int kleinerAnzahl = 0;
  int gleichAnzahl = 0;
  int groesserAnzahl = 0;
  for (int i = 0; i < feld.length; i++) {
    if (feld[i] < schnittzahl)
      kleinerAnzahl++;
    else
      if (feld[i] == schnittzahl)
        gleichAnzahl++;
      else
        groesserAnzahl++;
  }

  // Initialisierung der Teilfelder
  if (kleinerAnzahl > 0)
    kleinerFeld  = new int[kleinerAnzahl];
  gleichFeld   = new int[gleichAnzahl];
  if (groesserAnzahl > 0)
    groesserFeld = new int[groesserAnzahl];

  // Zerlegung von feld
  int kleinerIndex = 0;
  int gleichIndex = 0;
  int groesserIndex = 0;
  for (int i = 0; i < feld.length; i++) {
    if (feld[i] < schnittzahl)
      kleinerFeld[kleinerIndex++] = feld[i];
    else
      if (feld[i] == schnittzahl)
        gleichFeld[gleichIndex++] = feld[i];
      else
        groesserFeld[groesserIndex++] = feld[i];
  }

  // 3. Schritt: Rekursive Aufrufe zur Sortierung der Teilfelder
  kleinerFeld  = sortiere(kleinerFeld);
  groesserFeld = sortiere(groesserFeld);
```

```
  // 4. Schritt: Zusammenfuegen des Ergebnisses
  int i = 0;
  for (int j = 0; j < kleinerAnzahl; j++)
    ergebnis[i++] = kleinerFeld[j];
  for (int j = 0; j < gleichAnzahl; j++)    // i nicht zuruecksetzen!
    ergebnis[i++] = gleichFeld[j];
  for (int j = 0; j < groesserAnzahl; j++)  // i nicht zuruecksetzen!
    ergebnis[i++] = groesserFeld[j];

  // 5. Schritt: Aufruf beenden (Rückgabe des sortierten Feldes)
  return ergebnis;
}
```

Eine kürzere Variante ist die Einbettung der bereits vorhandenen prozeduralen `sortiere`-Methode in eine funktionale Methode `sortiere2`. Dazu muß lediglich das Eingabefeld kopiert werden. Als Ergebnis wird dann die sortierte Kopie zurückgeliefert:

```
// Rekursive Sortierung mit Einbettung
static int[] sortiere2 (int[] feld) {
  // Kopieren des Eingabefelds
  int[] ergebnis = new int[feld.length];
  for (int i = 0; i < feld.length; i++)
    ergebnis[i] = feld[i];

  sortiere(ergebnis);
  return ergebnis;
}
```

c) Eine Testklasse für die zuvor erstellten Methoden könnte z.B. folgendermaßen deklariert werden:

```
// SortierTest:
// Testen der rekursiven Sortier-Methoden
//
class SortierTest {
  // Klassenweite Deklaration des Testfelds
  static int[] testFeld = { 9, 7, 3, 4, 5, 6, 8, 0, 1, 2 };

  // Prozedurale Methode fuer Sortier-Algorithmus
  static void sortiere (int[] feld) {...}     // Methodenrumpf siehe a)

  // Funktionale Methode fuer Sortier-Algorithmus
  static int[] sortiere2 (int[] feld) {...}   // Methodenrumpf siehe b)

  // main-Methode zum Testen der Sortier-Methoden
  public static void main (String[] args) {
    // Ausgabe des unsortierten Feldes
    System.out.print("Testfeld:           ");
    for (int i = 0; i < testFeld.length; i++)
```

```
      System.out.print(" " + testFeld[i]);
    System.out.println("");

    // Zuerst die funktionale Variante testen (testFeld bleibt unsortiert)
    int[] ergebnis = sortiere2(testFeld);
    System.out.print("Funktional sortiert:");
    for (int i = 0; i < ergebnis.length; i++)
      System.out.print(" " + ergebnis[i]);
    System.out.println("");

    // Erst jetzt testfeld selbst sortieren (prozedural)
    sortiere(testFeld);
    System.out.print("Prozedural sortiert:");
    for (int i = 0; i < testFeld.length; i++)
      System.out.print(" " + testFeld[i]);
    System.out.println("");
  }
}
```

Der vollständige Programmtext ist in der Datei

`/usr/proj/javakurs/Tag1/loesungen/SortierTest.java`

abgelegt.

Aufgabe 5 Zugriffe auf Felder

a) Wird der "=="-Operator auf zwei Felder angewandt, so wird überprüft, ob es sich bei beiden Feldern um ein und dasselbe Feld handelt oder nicht. Es wird also überprüft, ob die zwei verglichenen Feldreferenzen (Felder gehören zu den Referenztypen!) auf denselben Speicherbereich verweisen.

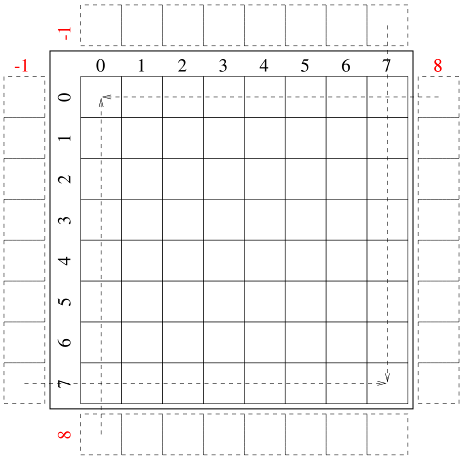
Soll dagegen überprüft werden, ob zwei unterschiedliche Felder inhaltlich identisch sind, müssen die einzelnen Feldelemente paarweise (z.B. in einer `for`-Schleife) auf Gleichheit überprüft werden. Der `==`-Operator, wenn er auf die Referenzen angewendet wird, ist dafür nicht ausreichend. Dies gilt analog für alle Referenztypen, also auch für Objekttypen, wie sie im weiteren Kursverlauf vorgestellt werden.

b) Zur Repräsentation des Spielbretts bietet sich in Java ein zweidimensionales Feld an (Zeilen und Spalten), wobei die Feldlänge für beide Dimensionen jeweils mit 8 initialisiert wird. Der Datentyp der einzelnen Feldelemente (entsprechend einem Feld des Spielbretts) ist für diese Aufgabe nicht von Bedeutung.

Eine Bewegung erfolgt durch Addition bzw. Subtraktion (je nach Bewegungsrichtung) der jeweiligen Schrittzahlen auf die Zeilen- und Spaltenindizes, die der aktuellen Position der Spielfigur entsprechen:

```
// aktuelle Position sei in positionZeile und positionSpalte gesichert
// feldInhalt vom gleichen Datentyp wie die Feldelemente
// Sprung 2 nach links, 1 nach oben:
feldInhalt = spielbrett[positionZeile - 1][positionSpalte - 2];
```

Entscheidend für den korrekten Zugriff auf Feldelemente ist die Einhaltung der Indexgrenzen. Für jede Dimension muß der jeweilige Index stets zwischen 0 und dem für diese Dimension zulässigen maximalen Indexwert liegen. Für unser Spielbrett bedeutet das, daß nur Indexwerte zwischen 0 und 7 zulässig sind.

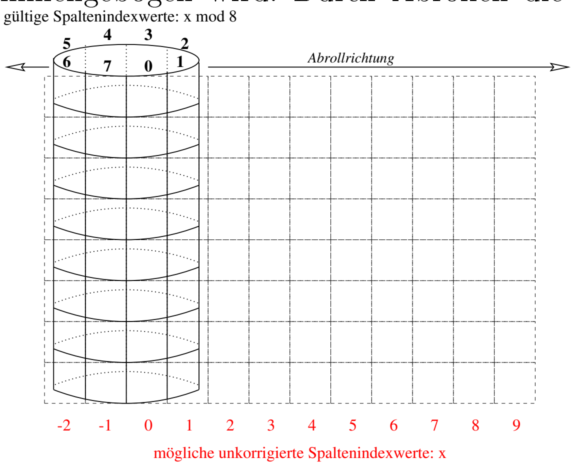


Werden durch die Bewegung der Figur die Indexgrenzen über- bzw. unterschritten, so muß eine Korrektur erfolgen. Wird z.B. die obige Bewegung von Position `[0][0]` aus ohne Korrektur durchgeführt, so ergibt sich als Position `[-1][-2]`, was innerhalb eines Programmablaufs unweigerlich zu einer Fehlermeldung führt. Korrekt wäre `[7][6]` als neue Position.

Wir benötigen also eine Abbildungsvorschrift, die alle möglichen Werte für den Zeilen- bzw. Spaltenindex, die durch eine Bewegung der Spielfigur auftreten können, eindeutig auf eine existierende Zeile bzw. Spalte abbildet.

Die Indexberechnungen für Zeilen und Spalten können unabhängig voneinander durchgeführt werden. Zur Vereinfachung entwickeln wir im folgenden nur die Korrektur für den Spaltenindex. Das dargestellte Prinzip gilt aber analog auch für die Indizierung von Zeilen.

Zur Veranschaulichung können wir uns vorstellen, daß das Spielbrett (bei Betrachtung des Spaltenindexes) vertikal zu einen Zylinder zusammengebogen wird. Durch Abrollen dieses Zylinders über den Wertebereich der unkorrigierten Indexwerte können wir für jeden dieser Werte den korrekten Spaltenindex ermitteln.



Die nötige Korrekturberechnung erfolgt mit Hilfe der *Modulo*-Operation ($x \bmod n$), die eine beliebige ganze Zahl x auf einen Wert aus dem Bereich $0 \ldots (n-1)$ abbildet. Die folgende Berechnungsvorschrift definiert die Modulo-Operation rekursiv:

$$\mathrm{mod} : Z \times N \longrightarrow N_0$$

$$x \bmod n = \begin{cases} (x-n) \bmod n, & \text{falls} \quad x \geq n \\ x, & \text{falls} \quad 0 \leq x \leq n-1 \\ (x+n) \bmod n, & \text{falls} \quad x < 0 \end{cases}$$

Ein Rekursionsschritt kann nach dieser Definition als eine komplette Umdrehung des Spielbrettzylinders interpretiert werden. Das Abrollen erfolgt dann, ausgehend von der 0, in Richtung auf den Indexwert x.

Eine nicht-rekursive Definition der Modulo-Operation sieht folgendermaßen aus:

$$\mathrm{mod} : Z \times N \longrightarrow N_0$$

$$x \bmod n = \begin{cases} x - n \cdot \left\lfloor \frac{x}{n} \right\rfloor, & \text{falls} \quad x \geq n \\ x, & \text{falls} \quad 0 \leq x \leq n-1 \\ x - n \cdot \left\lceil \frac{x}{n} \right\rceil, & \text{falls} \quad x < 0 \end{cases}$$

Wie man sieht, entspricht die Modulo-Operation für nicht-negative Werte der (ganzzahligen) Restbildung, die in Java durch den `%`-Operator ausgedrückt wird. Für negative Werte (in unserem Fall sind nur -1 oder -2 theoretisch möglich) wird entsprechend der oben angegebenen Berechnungsvorschrift n (in unserem Beispiel also 8) addiert.

Eine Methode zur korrekten Indexberechnung könnte damit beispielsweise so aussehen:

```
// Modulo-Abbildung von Indexwerten zwischen -2 und 9
int indexModulo (int index) {
  if (index < 0)
    index += 8;
  return index % 8;
}
```

Diese Methode kann genauso für die Korrektur von Zeilenpositionen verwendet werden, so daß die neue Position der Spielfigur nach einer Sprungbewegung folgendermaßen ermittelt werden kann:

```
// aktuelle Position sei in positionZeile und positionSpalte gesichert
// feldInhalt vom gleichen Datentyp wie die Feldelemente
// Sprung 2 nach links, 1 nach oben:
feldInhalt = spielbrett[indexModulo(positionZeile - 1)]
                       [indexModulo(positionSpalte - 2)];
```