Schleifen in Java

In Java gibt es vier Arten von Schleifen

1. Abweisende Schleifen (while)
2. Akzeptierende Schleifen (do ... while)
3. Zählschleifen (for)
4. Iterator-Schleifen (for)

Abweisende Schleifen kennen wir schon, die anderen folgen jetzt nacheinander

Motivation: Akzeptierende Schleifen

- Aufgabe:
  Abfrage, ob ein Programmteil wiederholt werden soll;
  erlaubt sind nur die Eingaben 'j' und 'n' für ja bzw. nein
- Lösung mit einer (abweisenden) *while*-Schleife

```
...
String eingabe;
print("Wollen Sie den Vorgang wiederholen (j/n): ");
eingabe = readString();
while ( !"j".equals(eingabe) && !"n".equals(eingabe) ) {
    print("Wollen Sie den Vorgang wiederholen (j/n): ");
    eingabe = readString();
}
...
```

Identischer Programmcode

- Der Programmcode muss wiederholt werden, weil der Rumpf einer abweisenden Schleife eventuell nie ausgeführt wird
- Günstiger wäre hier eine (akzeptierende) Schleife, deren Rumpf *auf jeden Fall mindestens einmal* ausgeführt wird

Ablauf akzeptierender Schleifen

Ablauf

1. Der Schleifenrumpf (Anweisung 1) wird ausgeführt
2. Die Bedingung wird ausgewertet
   - Ist die Bedingung erfüllt, dann wird das Programm mit Anweisung 1 fortgesetzt; der Schleifenrumpf wird erneut durchlaufen
   - Ist die Bedingung nicht erfüllt, dann wird das Programm mit Anweisung 2 fortgesetzt

Unabhängig von der Bedingung wird der Schleifenrumpf mindestens ein Mal durchlaufen

Akzeptierende Schleifen

Es sind drei Fälle möglich

- Bereits die erste Auswertung des Ausdrucks liefert *false*; die Schleife wurde zu diesem Zeitpunkt bereits ein Mal ausgeführt!
- Die wiederholte Auswertung des Ausdrucks liefert eine endliche Folge von
  true, *true*, ..., *true*, *false*
  von Ergebnissen
- Die wiederholte Auswertung des Ausdrucks ergibt immer den Wert *true* (*Endlosschleife*)

Bessere Lösung der Aufgabenstellung

- Bisherige Lösung mit einer *abweisenden while*-Schleife

```
...
String eingabe;
print("Wollen Sie den Vorgang wiederholen (j/n): ");
eingabe = readString();
while (!"j".equals(eingabe) && !"n".equals(eingabe) ) {
      print("Wollen Sie den Vorgang wiederholen (j/n): ");
      eingabe = readString();
}
...
```

- Neue Lösung mit einer *akzeptierenden do*-Schleife

```
...
String eingabe;
do {
      print("Wollen Sie den Vorgang wiederholen (j/n): ");
      eingabe = readString();
} while ( !"j".equals(eingabe) && !"n".equals(eingabe) );
...
```

Beispiel für eine akzeptierende Schleife 1

```
i = 1;
do
        i++;
while ( i < 10 );
println( i );
```

Ausgabe:

Beispiel für eine akzeptierende Schleife 2

```
i = 1;
do {
        i++;
        println( i );
} while ( i < 10 );
println( "Ende" );
```

Ausgabe:

Motivation: Zählschleifen

- Aufgabe:
  Ausgeben aller Elemente eines Arrays

- Lösung mit einer *while*-Schleife

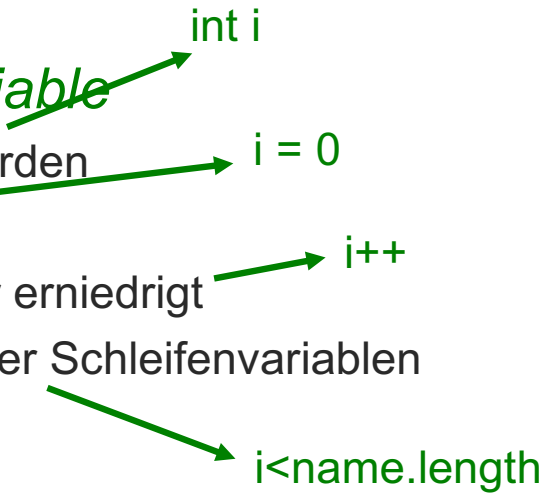
```
String [ ] name = new String[100];
int i;
...
i = 0;
while ( i<name.length ) {
    println( "Name an der Position " + i + ": " + name[i] );
    i++;
}
...
```

Solche Programmfragmente werden in Schleifen häufig benötigt

Daher gibt es für diesen Fall eine eigene Schleifenform, die *Zählschleife*

Schleifen in Algorithmen

- *Zählschleifen* besitzen eine *Schleifenvariable*
  - Diese kann in der Schleife selbst eingeführt werden
  - Sie wird in der Schleife initialisiert
  - Sie wird in der Schleife regelmäßig erhöht oder erniedrigt
  - Der Abbruch der Schleife wird über den Wert der Schleifenvariablen gesteuert




```
for ( int i = 0; i < name.length; i++ )
        println( ... );
```

Beispiel für eine for-Schleife 1/3

Initialisierung: Deklaration mit Initialisierung ***ODER*** Zuweisung

Bedingung:

- Muss vom Typ *boolean* sein
- Wenn sie nicht erfüllt ist, wird die Schleife beendet (abweisender Charakter)

Update: Zuweisung

```
for ( int i = 0; i < name.length; i++ )
        println( ... );
```

- Bemerkung
  Es ist durchaus möglich, Zählschleifen (fast vollkommen) anders zu verwenden (ein Beispiel wird noch gezeigt)
  Das ist aber ein sehr schlechter Programmierstil!
  In solchen Fällen sollten *immer* *while*- oder *do*-Schleifen verwendet werden

Beispiel für eine for-Schleife 2/3

```
for (i = 1; i <= 10; i++ )
        println( i );
```

Äquivalente *while*-Schleife

```
i = 1;
while ( i <= 10 ) {
        println( i );
        i++;
}
```

Beispiel für eine for-Schleife 3/3

2 Variablen, Initialisierungen **möglich**, aber nur ein Typ erlaubt

```
for ( int i = 10, j = 1; i >= 0; i--, j++ ) {
        println( „Schleifenvariable: “ + i );
        println( „Anzahl der Durchläufe: “ + j );
}
```

Beide Variablen **können** modifiziert werden

Deklarationen in for-Schleifen

```
{       int i;
        ...
        for ( int i = 10; i >= 0; i-- )
                ...
}
```

```
for ( int i = 10; i >= 0; i-- ) {
        ...
}
println( i );
```

```
for ( int i = 10; i >= 0; i-- ) {
        ...
}
int i = 5;
println( i );
```

Beispiele für „entfremdete“ for-Schleifen 1/3

- Folgende Schleifen sind (gültige) Endlos-Schleifen

```
for ( ; true; )
        println( "forever" );

for ( ; ; )
        println( "forever" );
```

Steht Programmtext nach einer dieser Schleifen, meldet der Compiler einen Fehler, weil dieser Programmteil nie erreicht werden kann

→ In beiden Fällen wäre eine *while*-Schleife angemessen:

```
while ( true )
        println( "forever" );
```

Beispiele für „entfremdete“ for-Schleifen 2/3

- Folgende Schleife wird „künstlich“ beendet

```
for ( int i = 1; i <= 100; i++ ) {
    ...
    if ( bedingung )
        break;
    ...
}
```

Sehr schlecht!!

Man übersieht leicht, dass die Schleife eventuell weniger als 100 Durchläufe hat

Beispiele für „entfremdete“ for-Schleifen 3/3

- Auch folgende Schleife wird „künstlich“ beendet

```
for ( int i = 1; i <= 100; i++ ) {
    ...
    if ( bedingung )
        i = 101;
    ...
}
```

Sehr sehr schlecht!!

Man übersieht leicht, dass die Schleife eventuell weniger als 100 Durchläufe hat

→ Abhilfe: Boole'sche Hilfsvariable verwenden, dann sieht man der Schleife an, dass ein anderer Abbruch möglich ist:

```
boolean elementGefunden = false;

for ( int i = 1; i <= 100 && !elementGefunden; i++ ) {
    ...
    elementGefunden = bedingung;
    ...
}
```

Motivation: Iterator-Schleifen

- Aufgabe:
  Aufsummieren, Minimum und Maximum *aller Elemente* eines Arrays bestimmen
- Lösung mit einer *while*-Schleife

```
int [ ] noten = new int[100];
int summe = 0, beste = 7, schlechteste = 0;
...
int i = 0;
while ( i < noten.length ) {
    summe += noten[i];
    if (noten[i] < beste)
            beste = noten[i];
    if (noten[i] > schlechteste)
            schlechteste = noten[i];
    i++;
}
...
```

Solche Programmfragmente werden in Schleifen besonders häufig benötigt

Daher gibt es für diesen Fall eine eigene Schleifenform, die *Iterator-Schleife*

Iteratoren in Algorithmen

- Iteratoren besitzen eine *automatisch kontrollierte Schleifenvariable*
  - Diese wird *vom Compiler* in der Schleife selbst hinzugefügt
  - Sie wird automatisch von der VM zu Beginn der Schleife initialisiert
  - Sie wird automatisch von der VM am Ende der Schleife auf das nächste Element gesetzt
  - Der Abbruch der Schleife ist erreicht, wenn mit jedem Array-Element ein Schleifendurchlauf durchgeführt wurde

Verschiedene Schleifen für die gleiche Aufgabe

- Aufgabe: Aufsummieren, Minimum und Maximum aller Elemente eines Arrays bestimmen
- Lösung mit einer *while*-Schleife

```
int [ ] noten = new int[100];
int summe, beste, schlechteste;
... // Initialisierungen!
int i = 0;
while ( i < noten.length ) {
      summe += noten[i];
      if (noten[i] < beste)
                  beste = noten[i];
      if (noten[i] > schlechteste)
                  schlechteste = noten[i];
      i++;
}
...
```

- Kürzere Lösung mit einer Zählschleife

```
for ( int i = 0; i < noten.length; i++ ) {
      summe += noten[i];
      if (noten[i] < beste)
            beste = noten[i];
      if (noten[i] > schlechteste)
            schlechteste = noten[i];
}
...
```

- Elegante Lösung mit einer Iterator-Schleife

```
for ( int note : noten ) {
      summe += note;
      if (note < beste)
                  beste = note;
      if (note > schlechteste)
                  schlechteste = note;
}
```

Vor- und Nachteile der Iterator-Schleife

+ Man macht keine Index-Fehler mehr
+ Man muss sich keine Indexvariable ausdenken, initialisieren und erhöhen, die nur als Mittel zum Zweck (= Zugriff auf die Array-Elemente, mit denen man arbeiten muss) dient

– Es geht nicht alles, manchmal braucht man die Indexvariable, z.B.
  – bei der gleichzeitigen Iteration über zwei Arrays,
  – beim Zugriff auf benachbarte Elemente, …