HAW Hamburg
Fakultät TI
Technik und Informatik

Programmiermethodik 1
Programmiertechnik

Schleifen

Änderungshistorie

- 14.4.15: Korrektur einiger Typos

Wiederholung

- Bedingte Anweisungen
  - if, else
  - dreistelliger bedingter Operator
  - switch
- Debugging
  - Tracing
  - Debugging in Eclipse

Ausblick für heute

Use Cases

- Wiederhole eine Anweisung bis eine vordefinierte Bedingung erreicht ist.
- Wiederhole eine Anweisung für eine vordefinierte Anzahl von Durchläufen.

Agenda

- while-Schleife
- do-while-Schleife
- break & continue
- Sichtbarkeitsbereiche
- for-Schleife

while-Schleife

Einführung

- Computer sind sehr gut darin,
  - ... etwas schnell zu machen.
  - ... etwas oft zu machen.
- häufige Anforderung
  - wiederholte Durchführung der gleichen (oder einer ähnlichen) Berechnung
  - Abbruch erst, wenn eine Bedingung erfüllt ist
    - "Abbruchkriterium"
- wiederholte Anwendung: Schleife
- in Java: verschiedene Schleifen-Typen
  - wieder: syntaktischer Zucker, alle Schleifen lassen sich ineinander überführen

while-Schleife

- Anwendungsfall
  - Wiederhole etwas, so lange eine Bedingung erfüllt ist
  - meist unbekannte Anzahl von Durchläufen
- Beispiel: Suche nach dem ASCII-Index für den Buchstaben 'a'.

```
/**
 * Suche den (ASCII)-Index des Zeichens 'a'.
 */
public class FindeZeichenIndex {

  /**
   * Programmeinstieg.
   */
  public static void main(String[] args) {
    System.out
        .println("Suche nach (ASCII)-Index des Buchstabens 'a' ...");
    int zeichenIndex = 0;
    while ((char) zeichenIndex != 'a') {
      zeichenIndex += 1;
    }
    System.out
        .format(
            "Der Buchstabe 'a' hat den Code %d.\n",
            zeichenIndex);
  }
}
```

while-Schleife

- Syntax

```
while (<Bedingung>) <Anweisung>
```

- sprich:
  - "Solange <Bedingung> gilt, wiederhole <Anweisung>
- Die Bedingung (logischer Ausdruck) steuert den Ablauf:
  - 1. Bedingung auswerten
  - 2. Falls …
    - `true`: Anweisung(en) ausführen und anschließend zurück zu 1.
    - `false`: `while`-Schleife beendet, nach der `while`-Schleife weiter

GGT-Algorithmus Euklid

- Euklids Algorithmus zum Berechnen des GGT (Größter gemeinsamer Teiler zweier Ganzzahlen)
- Algorithmus (Pseudocode)

```
Eingabe: zahl1, zahl2 (Ganzzahlen)
Ausgabe: ergebnis (Ganzzahl)
wenn zahl1 = 0
        dann ergebnis ← zahl2
sonst solange zahl2 ≠ 0
        wenn zahl1 > zahl2
                dann zahl1 ← zahl1 – zahl2
        sonst zahl2 = zahl2 – zahl1
ergebnis ← zahl1
```

Inkrement- und Dekrementoperator

- Schleifenvariablen werden oft in Einerschritten nach oben oder unten gezählt:
  - `variable = variable + 1;`
  - `variable = variable - 1;`
- spezielle unäre Operatoren erhöhen bzw. erniedrigen eine numerische Variable um 1 (Berechnung und Zuweisung):
  - Inkrementoperator: `++`
  - Dekrementoperator: `--`
- Anwendung:
  - `variable++;`
  - `variable--;`
- folgende Anweisungen sind (fast) äquivalent:
  - `variable = variable + 1; // nicht bei byte, char, short`
  - `variable++;`
  - `variable += 1;`

Geschachtelte Schleifen

- Eine `while`-Schleife ist selbst eine Anweisung
  - kann im Rumpf einer weiteren Schleife stehen: Geschachtelte Schleifen
- Beispiel:

```
int i = 1;
int j;
while(i <= 10) {                // äußere Schleife
        j = 1;
        while(j <= 10) {        // innere Schleife
                System.out.format("%4d", i*j);
                j++;
        }
        i++;
        System.out.println(); // Neue Zeile
}
```

Übung: while-Schleife

- Schreiben Sie ein Programm `Wuerfeln`. Das Programm soll so lange eine neue Zahl würfeln, bis eine "6" gewürfelt wurde. Simulieren Sie einen sechsseitigen Würfel.
- Hinweis: Mit `Math.random()` erzeugen Sie eine Zufallszahl aus dem Intervall [0;1[

do-while-Schleife

nochmal Würfeln

- Nachteil der Variante mit `while`
  - bereits außerhalb der Schleife muss einmal gewürfelt werden
- Alternative
  - `do-while`-Schleife

```
/**
 * Würfeln von W6-Zufallszahlen, bis eine 6 gewürfelt wurde.
 */
public class WuerfelnDoWhile {

  /**
   * Programmeinstieg.
   */
  public static void main(String[] args) {
    int wurf;
    do {
      wurf =
          (int) (6 * Math.random()) + 1;
      System.out.format("Wurf: %d.\n",
          wurf);
    } while (wurf != 6);
  }

}
```

do-while-Schleife

- Syntax

```
do <Anweisung> while (<Bedingung>);
```

- sprich: "Mache <Anweisung> solange wie <Bedingung> gilt"
- Die Bedingung (logischer Ausdruck) steuert den Ablauf:
  - 1. Anweisung(en) ausführen
  - 2. Bedingung auswerten
  - 3. Falls …
    - `true`: zurück zu 1.
    - `false`: Schleife beendet, nach der `do-while`-Schleife weiter

do-while-Schleife

- eine `do-while`-Schleife wird auf jeden Fall mindestens 1-mal durchlaufen
  - eine `while`-Schleife eventuell überhaupt nicht

Übung: Do-While-Schleife

- Schreiben Sie ein Java-Programm, das vom Benutzer wiederkehrend Zeichen abfragt, bis er/sie `'e'` eingibt. Dann soll das Programm enden.
- Ein einzelnes Zeichen können Sie durch den Scanner so einlesen:

```
scanner.next().charAt(0)
```

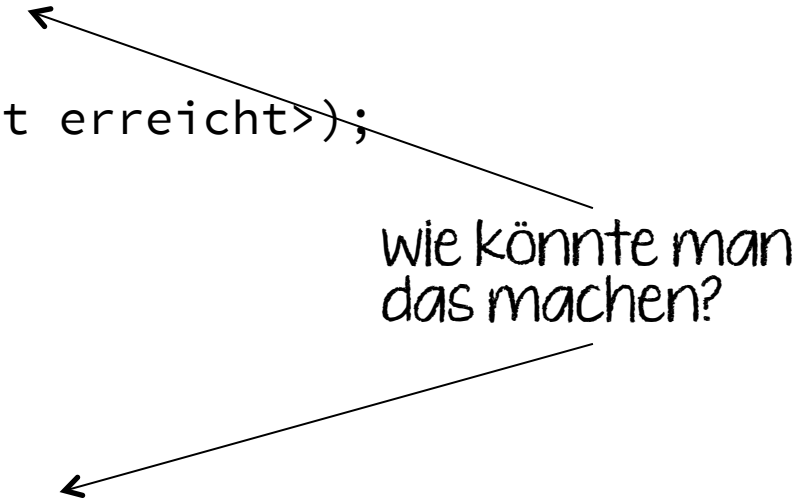
break & continue

Vorzeitiges Schleifenende


```
do {
        <Eingabe einlesen>
        <Falls Eingabe ungültig -> nächster Schleifendurchlauf>
        <Eingabe verarbeiten>
} while (<Abbruchbedingung nicht erreicht>);
```

oder

```
do {
        <Eingabe einlesen>
        <Falls Eingabe ungültig -> Beenden der Schleife>
        <Eingabe verarbeiten>
} while (<Abbruchbedingung nicht erreicht>);
```



break und continue

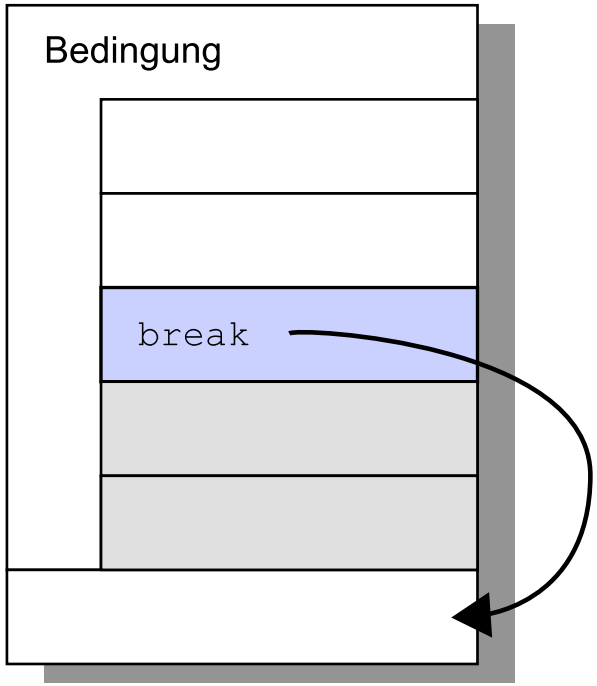
- unterbrechen des normalen Ablaufs von Schleifen
  - `break` und `continue`
- Im Rumpf von Schleifen zulässig
  - außerdem `break` in `switch`-Anweisungen
- Zweck:
  - übersichtlichere Formulierung von Schleifen
  - nützlich zur Behandlung von Sonderfällen
- aber: stehen der strukturierten Programmierung eigentlich entgegen
  - Kontrollfluss wird gespalten
  - vorsichtig einsetzen!
  - `break` und `continue` können grundsätzlich auch durch if-Anweisungen ersetzt werden!

Schleifenabbruch


```
int n;
while (true) {
        n = ...;  // Eingabe
        if (n == 0){
                break; // Ende
        }
        /* n verarbeiten */
        ...
}
```

Schleifenabbruch

- Auftauchen einer `break`-Anweisung innerhalb einer Schleife
  - sofortiges Verlassen der Schleife
  - Fortsetzung des Programms mit der ersten Anweisung nach der Schleife



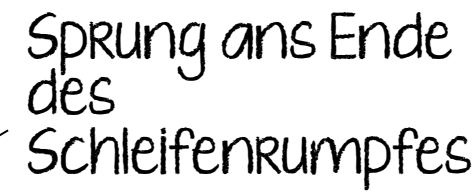
Übung: break

- Schreiben Sie ein Programm `EndeMitZahl`. Darin soll es eine "Endlos-While-Schleife" geben (betrachtet man nur die Abbruchbedingung).
- In jedem Schleifendurchlauf wird eine Ganzzahl eingelesen.
- Wird die Zahl 23 eingelesen, soll die Schleife abbrechen (also doch nicht endlos).

continue

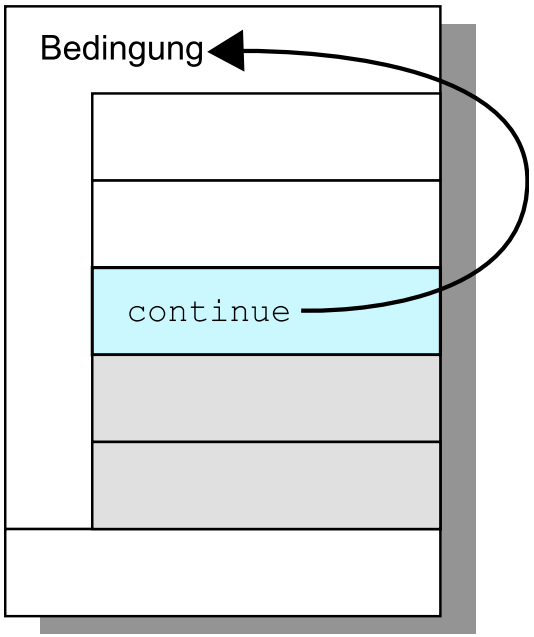

```java
/**
 * Berechnet die Summe aller geraden Zahlen aus Nutzereigaben.
 */
public class SummeGeraderZahlen {

  /**
   * Programmeinstieg.
   */
  public static void main(String[] args) {
    Scanner scanner =
        new Scanner(System.in);
    int zahl;
    int summe = 0;
    do {
      System.out
          .println("Bitte nächste Zahl eingeben, 0 für Ende.");
      zahl = scanner.nextInt();
      if (zahl % 2 == 1) {
        continue;
      }
      summe += zahl;
    } while (zahl != 0);
    System.out.println("Summe: "
        + summe + ".");
    scanner.close();
  }
}
```



continue

- Anweisung `continue` startet sofort den nächsten Schleifendurchlauf
  - der Rest des Rumpfes wird übersprungen



Übung: continue

- Schreiben Sie ein Programm `TeileZahl`.
- Das Programm teilt eine Zahl immer weiter durch je eine (Zufalls-)Zahl aus dem Intervall [-2, -1, ..., 2] (ganzzahlige Division).
- Das Programm endet, wenn die Zahl 0 erreicht.
- Nach jeder Division wird die Zahl auf der Konsole ausgegeben.
- Zu Beginn des Programms wird die Zahl mit 10 initialisiert
- Verhindern Sie Divisionen durch 0 mit Hilfe von `continue`.

Sichtbarkeitsbereiche

Sichtbarkeitsbereiche

- Blöcke gruppieren Anweisungen
- innerhalb eines Blocks sind alle Anweisungsarten erlaubt
  - auch Variablen-Deklarationen
- Gültigkeitsbereich (engl. scope) einer (lokalen) Variablen …
  - beginnt mit der Deklaration und
  - endet mit dem Block, in dem die Deklaration steht
- außerhalb des Blocks
  - Variable gilt nicht (ist nicht "sichtbar")
  - wird vom Compiler überprüft!

Sichtbarkeitsbereiche

- Beispiel
  - Deklaration von `j` nun innerhalb der while-Schleife:
  - `j` ist nur innerhalb der äußeren Schleife gültig
  - `i` auch außerhalb der äußeren while-Schleife

```
int i = 1;
while(i <= 10) {                        // äußere Schleife
        int j = 1;
        while(j <= 10) {                // innere Schleife
                System.out.format("%4d", i*j);
                j++;
        }
        i++;
        System.out.println(); // Neue Zeile
}
System.out.println(i);
// System.out.println(j); // Error
```

Namenskollisionen

- Gültigkeitsbereich einer lokalen Variablen umfasst untergeordnete (geschachtelte) Blöcke
- vorhergehendes Beispiel: `i` in beiden geschachtelten Blöcken verfügbar
- Namenskollision
  - Deklaration des gleichen Namens wie in einem übergeordneten Block

```
int i;
while(i <= 10) {
        int i;      // Namenskollision!
        …
}
```

- Java: doppelte Deklaration unzulässig!

> aber: keine Namenskollision in überschneidungs-freien Blöcken:

```
while(…) {
    int j;
    …
}
while(…) {
    int j;
    …
}
```

Lebensdauer einer Variablen

- Lebensdauer = Zeitintervall, in dem die Variable zur Laufzeit existiert
- die gleiche Variable wird möglicherweise ...
  - vielfach geschaffen ("Inkarnation")
  - zerstört (bei Verlassen des Gültigkeitsbereichs)
- aufeinander folgende Inkarnationen sind voneinander unabhängig
  - Beispiel 1×1-Programm:
    - `j` wird 10-mal geschaffen und 10-mal zerstört
- Schaffen und Zerstören (heute) praktisch ohne Laufzeitkosten

for-Schleife

Abbruchkriterium bei Schleifen

- offene Schleifen
  - Anzahl Schleifendurchgänge vorher nicht bekannt
  - Beispiele: GGT
  - Gefahr einer Endlosschleife!
- Zählschleifen
  - Anzahl Schleifendurchgänge liegt fest
  - Kontrolle mit einem „Schleifenzähler“

Schleifen mit fester Durchlaufzahl

- Beispiel: 1x1-Tabelle

```
for (int i = 1;  i <= 10;  i++){              // äußere Schleife
        for(int j = 1;  j <= 10;  j++){       // innere Schleife
                System.out.format("%4d", i*j);
        }
        System.out.println();
}
```

for-Schleife

- spezielle Schleife, optimiert für Zählvorgänge
- Syntax der `for`-Schleife („Für ...“):

```
for (<Start-Anweisung>; <Bedingung>; <Next-Anweisung>) <Anweisung>
```

- Ablauf:
  - 1. Start-Anweisung ausführen
  - 2. Bedingung auswerten
  - 3. Falls …
    - `true`: Anweisung(en) ausführen, danach Next-Anweisung ausführen und anschließend zurück zu 2.
    - `false`: `for`-Schleife beendet, nach der `for`-Schleife weiter

Übung: Zinsen

Erstellen Sie ein Programm `Zinsen` zur Zinsberechnung!

Anforderungsanalyse

- Eingabe
  - anzulegender Geldbetrag in EUR
    - Fließkommazahl
  - Zinssatz pro Jahr in Prozent
    - Fließkommazahl
  - Laufzeit in Jahren
    - ganzzahlig
- Ausgabe
  - für jedes Laufzeitjahr wird eine Zeile mit dem jeweiligen Gesamtwert der Geldanlage (inkl. Zins und Zinseszins) ausgegeben
- Hinweis: Der Gesamtwert der Geldanlage erhöht sich in jedem Jahr um die Jahreszinsen!

Wechsel zwischen Schleifentypen

- beide Codestücke zeigen dasselbe Verhalten!

```
for (int zaehler = 0; zaehler < 10; zaehler ++) {
        System.out.println(zaehler );
}
System.out.println("Ende!");

int zaehler = 0;
while (zaehler < 10) {
        System.out.println(zaehler );
        zaehler ++;
}
System.out.println("Ende!");
```

Gegenüberstellung

- Syntax der for-Schleife

```
for (<Start-Anweisung>; <Bedingung>; <Next-Anweisung>) <Anweisung>
```

- äquivalente while-Schleife:

```
{
        <Start-Anweisung>;
        while (<Bedingung>) {
                <Anweisung>
                <Next-Anweisung>;
        }
}
```

Übung: Schleifenumwandlung

- Verändern Sie folgendes Codefragment so, dass nur while-Schleifen verwendet werden:

```java
/**
 * Findet die nächst größere Primzahl nach einer Startzahl.
 */
public class FindeNaechstePrimzahl {

  /**
   * Programmeinstieg.
   */
  public static void main(String[] args) {
    boolean istPrimzahl = false;
    int zahl = 123;
    do {
      zahl++;
      istPrimzahl = true;
      for (int i = 2; i < zahl; i++) {
        if (zahl % i == 0) {
          istPrimzahl = false;
          break;
        }
      }
    } while (!istPrimzahl);
    System.out.println(zahl);
  }
}
```

Zusammenfassung

- while-Schleife
- do-while-Schleife
- break & continue
- Sichtbarkeitsbereiche
- for-Schleife