Einführung in die Informatik
Iterations

Konstruktion, Anwendungen, Varianten

Wolfram Burgard

Motivation

- Im vorangegangenen Kapitel haben wir mit der `while`-Schleife eine Form von **Wiederholungsanweisungen** für **Iterationen** kennen gelernt.
- In diesem Kapitel werden wir etwas systematischer beschreiben, wie man `while`-Schleifen formuliert.
- Darüber hinaus werden wir in diesem Kapitel weitere Anwendungen der `while`-Schleife kennen lernen.
- Schließlich werden wir ein alternatives Schleifen-Konstrukt betrachten.

Formulieren von Schleifen

- Gegeben sei eine Situation, in der wir in vielen Methoden einer Klasse eine Potenz für ganze Zahlen mit nicht-negativen, ganzzahligen Exponenten berechnen müssen:

$$x^y$$

- Der Prototyp einer solchen Methode ist offensichtlich

```
private int power(int x, int y)
```

- Da uns an Grundrechenarten die Multiplikation zur Verfügung steht, können wir die Potenz durch wiederholte Multiplikationen realisieren.

Informelle Beschreibung des Verfahrens

- Zur Formulierung des Verfahrens betrachten wir zunächst, wie wir die Berechnung von x^y per Hand durchführen würden:

$$x^y = \begin{cases} 1 & \text{falls } y = 0 \\ \underbrace{x * \ldots * x}_{y \text{ mal}} & \text{sonst} \end{cases} = 1 * \underbrace{x * \ldots * x}_{y \text{ mal}}$$

- Daraus ergibt sich ein informelles Verfahren:
  1. starte mit `1`
  2. multipliziere sie mit `x`
  3. multipliziere das Ergebnis mit `x`
  4. führe Schritt 3) solange aus, bis `y` Multiplikationen durchgeführt wurden.

Wahl und Definition der Variablen

- Der nächste Schritt ist, dass wir die Informationen bestimmen, die wir während unserer informellen Prozedur benötigt haben.
- Im einzelnen sind dies zwei Werte:
  1. Zunächst benötigen wir das **Ergebnis der zuletzt durchgeführten Multiplikation**.
  2. Darüber hinaus müssen wir **mitzählen, wie viele Multiplikationen wir bereits ausgeführt haben**.
- Wenn wir für das Ergebnis die Variable `result` und zum Zählen die Variable `count` verwendet, erhalten wir folgende Deklarationen:

```
int count,       // Anzahl durchgeführter Multiplikationen
    result;      // result == 1*x*...*x count mal
```

- Bisher sind die Variablen allerdings noch nicht initialisiert.

Das Skelett des Codes

Im nächsten Schritt formulieren wir ein Skelett des Codes einschließlich

1. des Methodenkopfes (Prototyp),
2. der Deklarationen und
3. des Skelettes der `while`-Schleife

```
private int power(int x, int y) {
    int count,      // Anzahl durchgeführter Multiplikationen
    result;         // result == 1*x*...*x count mal
    ...
    while (condition)
        body
    ...
}
```


Die Bedingung der while-Schleife (1)

- Um die **Bedingung der `while`-Schleife** zu formulieren, müssen wir wissen, **was nach Beendigung der `while`-Schleife gelten soll**.
- Dafür müssen wir wiederum festlegen, welchen Status unsere Variablen nach Beendigung der `while`-Schleife haben sollen und welche Operationen das Programm ausgeführt haben soll.
- In dem Fall der Potenzierung sind wir fertig, wenn wir y Multiplikationen durchgeführt haben, d.h. wenn der Wert von `count` mit dem von `y` übereinstimmt.
- Entsprechend dem Kommentar in der Deklaration, hat die Variable `result` stets den Wert x^{count} .
- Demnach hat `result` den Wert x^y, wenn `count` nach Beendigung der Schleife den Wert y hat.

Die Bedingung der while-Schleife (2)

- Ziel ist nun, dass `count` nach Beendigung der `while`-Schleife den gleichen Wert hat wie `y`.
- Da wir mit `count` die Anzahl der durchgeführten Multiplikationen zählen, müssen wir die Schleife solange wiederholen, bis `count` den gleichen Wert wie y hat.

```
private int power(int x, int y){
    int count,      // Anzahl durchgeführter Multiplikationen
        result;     // result == 1*x*...*x count mal
    ...
    while (count != y)
        body

    // count == y, result == x**y
    return result;
}
```

Initialisierung (1)

- Wenn das `while`-Statement erreicht wird, müssen die Variablen initialisiert sein, da andernfalls z.B. die Bedingung des `while`- Statements nicht sinnvoll ausgewertet werden kann.
- In unserem Beispiel müssen wir die Variable `count` entsprechend initialisieren.
- In unserer `while`-Schleife zählt `count` die Anzahl durchgeführter Multiplikationen.
- Deswegen sollte `count` vor Eintritt in die `while`-Schleife den Wert `0` haben.

Initialisierung (2)

- Unser Code muss daher folgendermaßen aussehen:

```
private int power(int x, int y){
    int count,      // Anzahl durchgeführter Multiplikationen
        result;     // result == 1*x*...*x count mal

    ...
    count = 0;
    while (count != y)
        body

    // count == y, result == x**y
    return result;
}
```


Der Rumpf der Schleife (1)

Bei der Formulierung des Rumpfes der Schleife müssen wir auf zwei Dinge achten:

1. Zunächst müssen wir garantieren, dass die Schleife terminiert bzw. stoppt. Der Rumpf einer Schleife wird nicht ausgeführt, wenn die Bedingung im Schleifenkopf zu `false` ausgewertet wird.
   Der Schleifenrumpf muss daher Anweisungen enthalten, die einen Fortschritt im Hinblick auf die Terminierung realisieren, d.h. Code, der dafür sorgt, dass die Schleifenbedingung irgendwann den Wert `false` hat.

   Unsere `while`-Schleife soll terminieren, sobald `count == y`.

Der Rumpf der Schleife (2)

Da wir in jedem Schleifendurchlauf eine Multiplikation ausführen und `count` die Anzahl notwendiger Multiplikationen zählt, müssen wir in jeder Runde `count` um `1` erhöhen.

```
private int power(int x, int y){
    int count,      // Anzahl durchgeführter Multiplikationen
        result;     // result == 1*x*...*x count mal
    ...
    count = 0;
    while (count != y){
        rest of body
        count++;
    }
    // count == y, result == x**y
    return result;
}
```

Der Rumpf der Schleife (3)

2. Darüber hinaus müssen wir Statements in den Rumpf einfügen, welche die erforderlichen Berechnungen durchführen.

   In unserem Beispiel zählt `count` die Anzahl durchgeführter Multiplikationen.

   Dementsprechend müssen wir in jedem Schleifendurchlauf auch eine Multiplikation ausführen, sodass `result` stets den Wert x^{count} hat.
   In unserem Beispiel wird das realisiert durch:

```
private int power(int x, int y){
    ...
    while (count != y){
        result *= x;
        count++;
    }
    ...
}
```


Akkumulatoren

- Die Variable `result` enthält in unserem Programm stets das aktuelle Zwischenergebnis x^{count} .
- In `result` speichern wir das aktuelle Teilprodukt.
- Daher nennen wir `result` einen **Akkumulator**.
- Akkumulatoren tauchen immer auf, wenn z.B. Summen oder Produkte über eine Sequenz von Objekten berechnet werden soll.
- Beispiele sind Summe der Einkommen über alle Mitarbeiter, Summe der Länge aller Zeilen in einem Text etc.

Initialisierung (erneut)

- Wie bereits erwähnt, müssen alle Variablen korrekt initialisiert sein, bevor wir in die Schleife eintreten.
- Von den beiden im Rumpf der Schleife vorkommenden Variablen haben wir `count` bereits korrekt initialisiert.
- In unserem Beispiel soll `result` stets den Wert x^{count} haben.
- Da `count` mit `0` initialisiert wird, müssen wir `result` den Wert $1 = x^0$ geben.

```
private int power(int x, int y){
    int count,      // Anzahl durchgeführter Multiplikationen
        result;     // result == 1*x*...*x count mal
    result = 1;
    count = 0;
    ...
    return result;
}
```


Die komplette Prozedur

```
static int power(int x, int y){
    int count,     // Anzahl durchgeführter Multiplikationen
       result;     // result == 1*x*...*x count mal
    result = 1;
    count = 0;
    while (count != y) {
        result *= x;
        count++;
    }
    // count == y, result == x**y
    return result;
}
```

Die einzelnen Schritte zur Formulierung einer Schleife

1. Aufschreiben einer informellen Prozedur
2. Bestimmen der Variablen
3. Das Skelett der Schleife
4. Die Bedingung der `while`-Schleife
5. Initialisierung der Variablen in der Bedingung
6. Erreichen der Terminierung
7. Vervollständigen des Rumpfes mit notwendigen Statements
8. Initialisierung der anderen im Rumpf benötigten Variablen

Muster einer Schleife: Zählen

- In der Praxis ist es häufig erforderlich, Objekte zu zählen.
- Typische Beispiele sind die Anzahl der weiblichen Studierenden, die Anzahl derer, die 50% der Übungsaufgaben gelöst haben etc.
- Um zu zählen, müssen wir entweder durch eine Kollektion laufen oder Objekte einlesen.
- Für das Zählen verwenden wir üblicherweise einen **Zähler**, d.h. eine Variable vom Typ `int`, die wir hier `count` nennen:

```
int count = 0;   //count == Anzahl der entsprechenden Fälle
...
while (...) {
    ...
    if (Objekt erfüllt bestimmte Bedingung)
    count++
    ...
}
```


Zählen der weiblichen Studierenden, welche mehr als 50 Punkte erreicht haben

1. Es gibt eine Klasse Student, mit zwei Methoden `isFemale` und `getPoints`.
2. Die Methode `isFemale` liefert genau dann `true`, wenn das entsprechende Objekt eine Studentin repräsentiert.
3. Die Methode `getPoints` liefert die erreichte Punktzahl.
4. Alle Studierenden sind in einem Kollektionselement `studentSet` zusammengefasst.

```
Enumeration e = studentSet.elements();
Student stud;
int count = 0;
while (e.hasMoreElements()){
    stud = (Student) e.nextElement();
    if (stud.isFemale() && stud.getPoints() > 50)
        count++;
}
```


Muster einer Schleife: Finden des Maximums

- Das Finden des maximalen Elements ist ebenfalls ein typisches Problem in einer Vielzahl von Anwendungen.
- Eine Beispielanwendung sind Wetterdaten: Welcher Tag war der heißeste im Jahrhundert?
- Da man das Auffinden des Maximums üblicherweise auf eine große Zahl von Objekten anwenden will, können wir davon ausgehen, dass die Objekte entweder eingelesen werden oder in einer Kollektionsklasse zusammengefasst sind.

Typisches Vorgehen zum Finden des Maximums

```
someType  extreme;  // extreme == Referenz auf ein Objekt,
                    // welches den maximalen Wert unter allen
                    // bisher betrachteten Elementen hat.
                    // extreme == null, wenn es kein Objekt
                    // gibt.

extreme = null;

while (...) {
    ...
    if (extreme == null || aktuelles Objekt ist größer
                           als alle bisher betrachteten)
    extreme = aktuelles Objekt;
}
```


Anwendungsbeispiel: Längste Zeile in einem Text

```
Enumeration e = v.elements();
String s;
String longest = null;

while (e.hasMoreElements()) {
    s = (String) e.nextElement();
    if (longest == null || s.length() > longest.length())
        longest = s;
}
System.out.println("Longest String is " + longest);
```


Maximaler Wert für primitive Datentypen

- Die Variante für primitive Datentypen ist sehr ähnlich zu der für Objekte.
- Allerdings enthalten primitive Datentypen wie `int` oder `float` nicht den Wert `null`.
- Deswegen müssen wir eine Variable vom Typ `boolean` verwenden, um zu repräsentieren, dass noch kein Wert betrachtet wurde.

Muster einer entsprechenden Schleife

```
someType  extreme;  // extreme == Der bisher gefundene
                    // Extremwert unter allen
                    // bisher betrachteten Elementen.
                    // Ist foundExtreme == false, ist der Wert
                    // bedeutungslos.
boolean foundExtreme;

foundExtreme = false;
extreme = beliebiger Wert;
while (...) {
    ...
    if (!foundExtreme || aktueller Wert ist größer
                         als alle bisher betrachteten){
        extreme = aktueller Wert;
        foundExtreme = true;
   }
}
```

Beispiel: Kleinste, von der Tastatur eingelesene Zahl

```
import java.io.*;

class ProgramSmallest {
    public static void main(String arg[]) throws Exception{
        BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
        boolean foundExtreme = false;
        int smallest = 0;
        String line;
        while ((line = br.readLine()) != null) {
            int x = Integer.parseInt(line);
            if (!foundExtreme || x < smallest){
                smallest = x;
                foundExtreme = true;
            }
        }
        if (foundExtreme)
            System.out.println("Smallest number is " + smallest);
   }
}
```


Die for-Schleife

- Speziell für Situationen, in denen die Anzahl der Durchläufe von Beginn an feststeht, stellt Java mit der `for`-Schleife eine Alternative zur `while`-Schleife zur Verfügung.
- Die allgemeine Form der `for`-Schleife ist:

```
for (Initialisierungsanweisung; Bedingung; Inkrementierung)
    Rumpf
```

- Sie ist äquivalent zu

```
Initialisierungsanweisung
while (Bedingung){
    Rumpf
    Inkrementierung
}
```


Potenzierung mit der for-Anweisung

- Bei der Potenzierung mussten wir genau y Multiplikationen durchführen.
- Die Anzahl durchgeführter Multiplikationen wurde in der Variablen `count` gespeichert.

```
static int power(int x, int y){
    int count, result = 1;

    for (count = 0; count < y; count++)
        result *= x;

    return result;
}
```


Komplexere for-Anweisungen

- Die Initialisierungs- und die Inkrementierungsanweisung können aus mehreren, durch Kommata getrennten Anweisungen bestehen.
- Betrachten wir die analoge `while`-Schleife, so werden die Initialisierungsanweisungen vor dem ersten Schleifendurchlauf ausgeführt.
- Auf der anderen Seite werden die Inkrementierungsanweisungen am Ende jedes Durchlaufs ausgeführt.
- Damit können wir auch folgende `for`-Anweisung zur Berechnung von x^y verwenden:

```
for (count = 0, result = 1; count < y; result*=x, count++);
```

- Solche **kompakten Formen der `for`-Anweisung** sind **üblicherweise schwerer verständlich** und daher **für die Praxis nicht zu empfehlen**.

Realisierung einer Endlosschleife mit der for-Anweisung

- Viele Systeme sind eigentlich dafür gedacht, permanent zu laufen.
- Typische Beispiele sind Web-Server oder Betriebssysteme.
- Da es für solche Systeme eher die Ausnahme ist, dass sie terminieren (es soll z.B. nur dann geschehen, wenn der Rechner ausgeschaltet werden soll), laufen sie üblicherweise in einer Endlosschleife.
- **Endlosschleifen** implementiert man auf **naive Weise** dadurch, dass man den Initialisierungs-, den Bedingungs- und den Inkrementierungsteil leer lässt, wobei das **Anhalten** dann beispielsweise durch ein return realisiert wird.
- In der Praxis gibt es Alternativen zu der hier vorgestellten Lösung mittels einer Endlosschleife. Diese haben den Vorteil, dass keine Rechenleistung verbraucht wird, während der Rechner z.B. auf Eingaben wartet.

Typisches Muster einer `for`-Endlosschleife

```
for (;;) {
    ...
    if (...) {
        ...
        return;
   }
   ...
}
```


Bedingte Auswertung logischer Ausdrücke

- Wie bereits erwähnt, wertet Java Ausdrücke von links nach rechts aus.
- Im Fall Boolescher Ausdrücke wertet Java jedoch nicht immer alle Teilausdrücke aus.
- Betrachten Sie folgenden Booleschen Ausdruck:

  `hours >= 40 && hours < 60`
- Hat `hours` den Wert 30, gibt es keinen Grund, den zweiten Vergleich `hours < 60` noch auszuwerten, weil der Ausdruck in jedem Fall `false` sein wird.
- Java bricht daher die Auswertung ab, sofern das Ergebnis bereits feststeht.
- Die bedingte Auswertung im Fall von `||` ist analog.

Ausnutzung der Bedingten Anweisung bei Tests

- Die Tatsache, dass Java logische Ausdrücke nur bedingt auswertet, haben wir bei folgendem Test ausgenutzt:

  `if (longest == null || s.length() > longest.length())`
- Falls auch im Fall `longest == null` beide Ausdrücke ausgewertet würden, so würde das zu einem Fehler führen, weil wir dann einer `null`-Referenz die Nachricht `length` schicken würden.

Zusammenfassung

- Um eine `while`-Schleife zu formulieren, geht man in verschiedenen Schritten vor.
- Für prototypische Problemstellungen, gibt es Muster einer Realisierung.
- Hierzu gehören das Zählen, die Akkumulation oder das Finden eines Maximums.
- Die `for`-Anweisung stellt eine Alternative zur `while`-Schleife dar.
- Logische Ausdrücke werden **bedingt** ausgewertet.

Terminologie

Akkumulator: Variable, die eine Teilsumme, ein Teilprodukt oder ein Teilergebnis einer anderen Operation als + und * enthält.

Zähler: Variable, die zum **Zählen** verwendet wird.

Iteration: Wiederholung einer Folge von Statements bis eine bestimmte Bedingung erfüllt ist.

Bedingte Auswertung: Logische Ausdrücke werden nicht weiter ausgewertet, sobald ihr endgültiger Wert bereits feststeht.